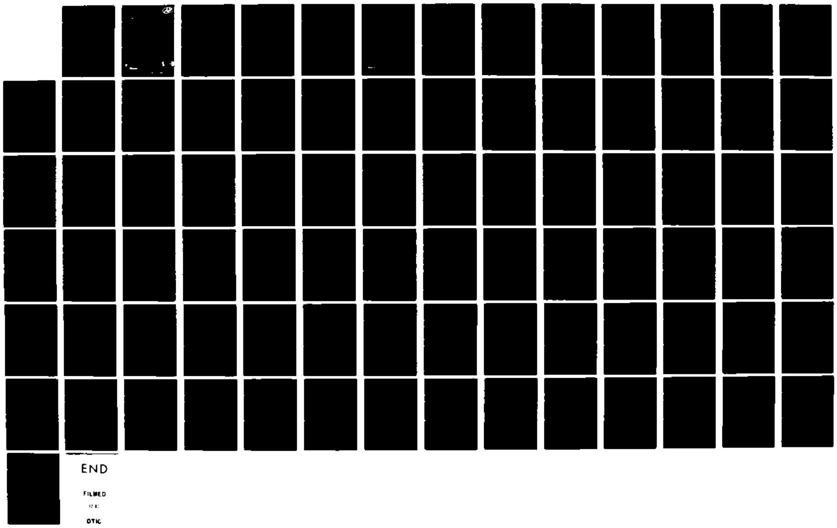
AD-A121 028 SWIRL: SIMULATING WARFARE IN THE ROSS LANGUAGE(U) RAND CORP SANTA MONICA CA P KLAHR ET AL. SEP 82 RAND/N-1885-AF F49620-82-C-0018 1/1

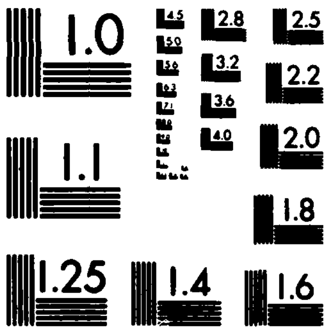
UNCLASSIFIED F/G 9/2 NL



END

FILMED

DTIC


1.0
4.5
5.0
5.6
6.3
7.1
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

82 11 09 002

The research reported here was sponsored by the Directorate of Operational Requirements, Deputy Chief of Staff/Research, Development, and Acquisition, Hq USAF, under Contract F49620-82-C-0018. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

Published by The Rand Corporation

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| | |
|---|---|
| **REPORT DOCUMENTATION PAGE** | READ INSTRUCTIONS BEFORE COMPLETING FORM |
| 1. REPORT NUMBER N-1885-AF / 2. GOVT ACCESSION NO. AD-A121028 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle) SWIRL: Simulating Warfare in the ROSS Language | 5. TYPE OF REPORT & PERIOD COVERED Interim |
| | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s) Philip Klahr, David McArthur, Sanjai Narain, Eric Best | 8. CONTRACT OR GRANT NUMBER(s) F49620-82-C-0018 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS The Rand Corporation 1700 Main Street Santa Monica, CA. 90406 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Requirements, Programs & Studies Group (AF/RDQM) Ofc, DCS/R&D and Acquisition Hq USAF, Washington, D.C. 20330 | 12. REPORT DATE September 1982 |
| | 13. NUMBER OF PAGES 73 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | 15. SECURITY CLASS. (of this report) Unclassified |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release: Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No Restrictions

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

| | |
|---|---|
| Computerized Simulation | Airborne Operations |
| Programming Languages | Attacks |

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

See Reverse Side

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SWIRL, a program for simulating military air battles, has evolved over the last two years as part of Rand's knowledge-based simulation research. This Note serves three purposes. First, it is a user's guide for those wishing to run SWIRL, and to some extent modify its behaviors. Second, the paper provides an extensive example of a simulator written in the ROSS language for those wishing to examine how one might design and build a simulator in ROSS. And third, at a more general level, the Note focuses on some important techniques for constructing simulations in an object-oriented programming environment.

# A RAND NOTE

# SWIRL: SIMULATING WARFARE IN THE ROSS LANGUAGE

Philip Klahr, David McArthur,
Sanjai Narain, Eric Best

September 1982

N-1885-AF

Prepared for The United States Air Force

Rand
SANTA MONICA, CA. 90406

## PREFACE

Under the sponsorship of Project AIR FORCE, Rand has been investigating techniques to significantly improve computer technology for military simulations and modeling. The project entitled "Computer Technology for Real-Time Battle Simulation" focuses on developing new technology to aid users in designing, building, understanding, and modifying battle simulations. This work led to the development of an English-like rule-oriented simulation language called ROSS (Rand Note N-1854-AF). To demonstrate the potential and payoff of the ROSS language, a prototype air penetration simulation, called SWIRL, was built. This Note describes the SWIRL design and implementation within the ROSS environment.

This Note is intended to serve three purposes. First, it is a user's guide to SWIRL. Those wishing to run SWIRL, and to some extent to modify its behaviors, can use this work as a guide. Second, the Note provides an extensive example of a simulation written in the ROSS language for those wishing to examine how one might design and build a simulation in ROSS. And third, at a more general level, it focuses on some important techniques for constructing simulations in an object-oriented programming environment.

## SUMMARY

Military battle simulations have been used extensively by the Air Force to study various open questions in such areas as mission planning and routing, force structure and employment, air basing, weapon systems, etc. Although such simulations have provided valuable information, they oftentimes lack the understandability and flexibility required by military planners and analysts. These simulations contain knowledge about the various objects being modeled including how the objects behave in various situations, how they interact with one another, and how they make decisions. Typically such knowledge is buried in unintelligible code that also lacks adequate documentation. Simulations would be even more valuable if users could better understand, and be able to modify, the behavioral knowledge embedded in such simulations. Analysts could then better experiment with alternative strategies and tactics for both offensive and defensive forces.

Applying and extending recent advances in computer science, artificial intelligence, and expert systems, we have been developing a prototype air battle simulation called SWIRL to demonstrate the payoff of the new technology to military simulations. In particular, SWIRL was designed to be fairly transparent so that users could readily read SWIRL code and easily make modifications. This task was accomplished primarily because of the English-like language used to build SWIRL. This language (ROSS -- Rule-Oriented Simulation System) was developed to be particularly suited for designing military battle simulations. It is described in detail in another report (Rand Note N-1854-AF).

## Summary

This Note describes the design and implementation of SWIRL and includes all of the SWIRL code and documentation. SWIRL embodies an air penetration simulation of offensive forces attacking a defensive area. Objects represented include offensive penetrators, defensive radars (both ground and air), SAMs, missiles, filter centers, defensive fighters, command centers and targets. A detailed example of a SWIRL simulation is provided.

## ACKNOWLEDGMENTS

Many persons have contributed to the development of SWIRL. Walter Mateskiela, Gary Massey, Russ Shaver, and Ross Quinlan helped develop various parts of SWIRL's knowledge base. William Giarla helped design and implement SWIRL graphics. And Henry Sowizral and Sally Goldin provided valuable comments and suggestions.

## TABLE OF CONTENTS

## 1.0. INTRODUCTION

Object-oriented programming languages such as SMALLTALK [GOL76], PLASMA [HEW77], and DIRECTOR [KAH79], as well as ROSS [MCA82], enforce a 'message-passing' style of programming. A program in these languages consists of a set of objects called actors that interact with one another via the transmission of messages. Each actor has a set of attributes and a set of message templates. Associated with each message template is a behavior that is invoked when the actor receives a message that matches that template. A behavior is itself a set of message transmissions to other actors. Computation is the selective invocation of actor behaviors via message passing.

This style of computation is especially suited to simulation in domains that may be thought of as consisting of autonomous interacting components. In such domains one can discern a natural mapping of their constituent components onto actors and of their interactions onto message transmissions. Indeed, experts in many domains may find the object-oriented metaphor a natural one around which to organize and express their knowledge [KLA80a]. In addition, object-oriented simulations can achieve high intelligibility, modifiability and credibility [FAU80, KLA80b, MCA81]. However, while these languages provide a potentially powerful simulation environment, they can easily be misused, since good programming style in object-oriented languages is not as well-defined as in more standard procedural languages.

In this paper we describe a program called SWIRL, designed for simulating military air battles. SWIRL has evolved over the last two years as part of the knowledge-based simulation research at Rand [FAU80, FAU81, KLA80a, KLA80b, GOL81, MCA81, MCA82]. The paper is intended to serve three purposes. First, it is a user's guide to SWIRL. Those wishing to run SWIRL, and to some extent modify its behaviors, can use this paper as a guide. Second, the paper provides an extensive example of a simulator written in the ROSS language [MAC82] for those wishing to examine how one might design and build a simulator in ROSS. And third, at a more general level, the paper focuses on some important techniques for constructing simulations in an object-oriented programming environment.

In the next section we discuss the goal of SWIRL, outline the main simulation objects comprising SWIRL, and note how they map onto objects in the air-battle domain. In Section 3 we briefly describe the computing environment in which we implemented SWIRL. In Section 4 we discuss the SWIRL user interface, i.e., how the user can modify SWIRL and get it to run specific simulations. In Section 5 we present detailed documentation of the SWIRL actors and their behaviors. Finally, in Section 6 we present the SWIRL code. The intent of the last two sections is to provide examples of good programming style in ROSS and to provide a resource for users of SWIRL who need to be able to modify the simulation code.

## 2.0. DESCRIPTION OF SWIRL

### 2.1. Goal

The goal of SWIRL is to provide a prototype of a design tool for military strategists in the domain of air battles. SWIRL embeds knowledge about offensive and defensive battle strategies and tactics. SWIRL accepts from the user a simulation environment representing offensive and defensive forces, and uses the specifications in its knowledge base to produce a simulation of an air battle. SWIRL also enables the user to observe, by means of a graphical interface, the progress of the air battle in time. Finally, SWIRL provides some limited user aids. Chief among these are an interactive browsing and documentation facility, written in ROSS, for reading and understanding SWIRL code, and an interactive history recording facility for analyzing simulation runs. This, coupled with ROSS's ability to easily modify simulation objects and their behaviors, encourages the user to explore a wide variety of alternatives in the space of offensive and defensive strategies and to discover increasingly effective options in that space.

### 2.2. Domain

In our air-battle domain, penetrators enter an airspace with a pre-planned route and bombing mission. The goal of the defensive forces is to eliminate those penetrators before they reach their targets. Below we list the objects that comprise this domain and briefly outline their behaviors. Figure 1 shows an example snapshot of an air-battle simulation.

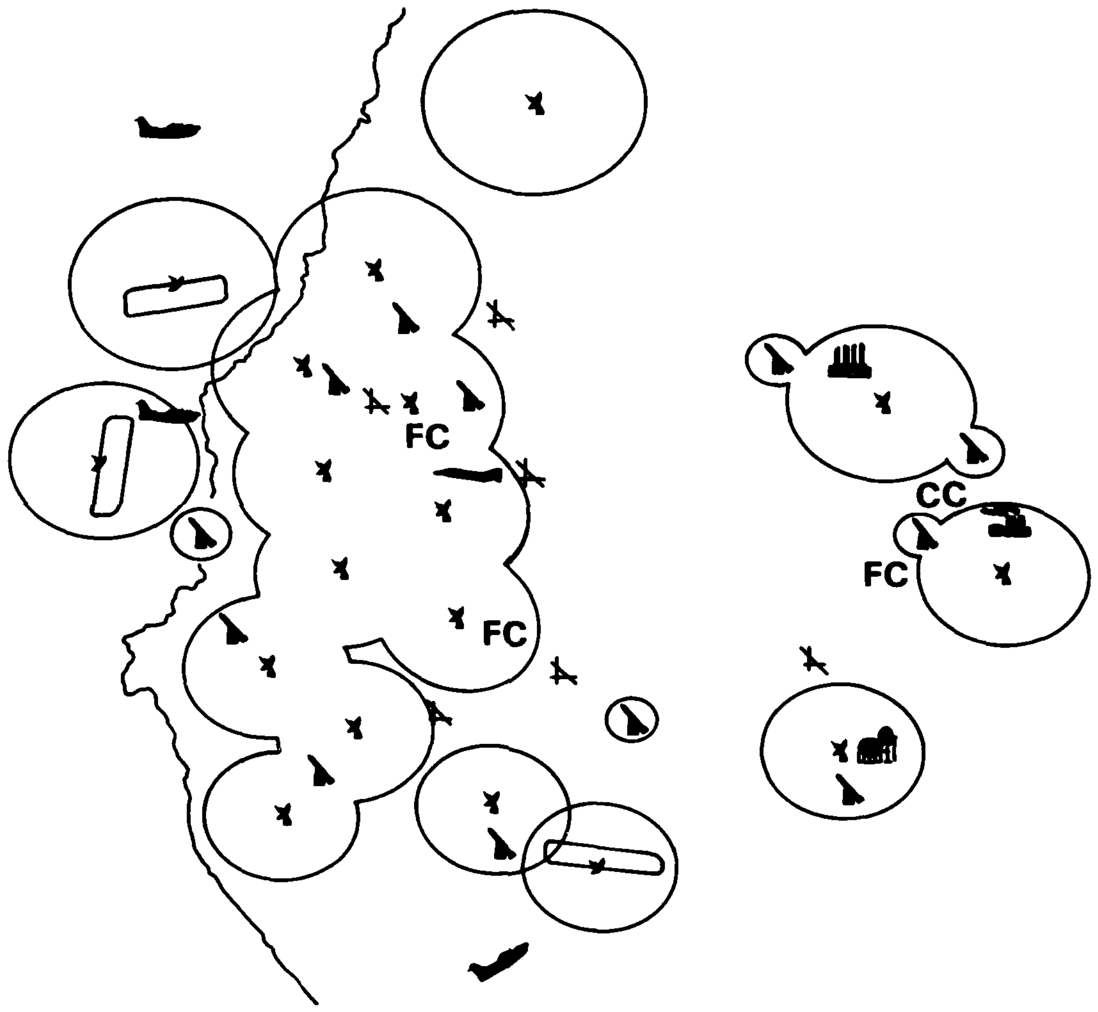


Figure 1. Graphical Snapshot of SWIRL Simulation

1. Penetrators. These are the primary offensive objects. They are assumed to enter the defensive air-space with a mission plan and route.

2. GCIs. Ground control intercept radars detect incoming penetrators and guide fighters to intercept penetrators.

3. AWACS. These are airborne radars that also detect and guide.

4. SAMs. Surface-to-air missile installations have radar capabilities and fire missiles at invading penetrators.

5. Missiles. These are objects fired by SAMs.

6. Filter Centers (FCs). They serve to integrate and interpret radar reports; they send their conclusions to command centers.

7. Fighter Bases. Bases are alerted by filter centers and send fighters out to intercept penetrators when requested to by command centers.

8. Fighters. Fighters receive messages from their base about their target penetrator. They are guided to the penetrator by a radar that is tracking the penetrator.

9. Command Centers (CCs). These represent the top-level in the command-and-control hierarchy. Command centers receive processed input about penetrators from filter centers and make decisions about which resource (fighter base) should be allocated to deal with a penetrator.

10. Targets. Targets are the objects penetrators intend to bomb.

## 2.3. Design

In this section we will outline how the above flow of command and control among the different kinds of objects is modeled in ROSS.

The first step in modeling in an object-oriented language such as ROSS is to decide upon the generic actors and their behaviors. A generic object or actor in ROSS represents an object type or class and includes the attributes and behaviors of all instances of that class. For example, the generic object 'fighter' represents each of the individual fighters that may be present in any simulation environment. Second, one may need to design a set of <u>auxiliary actors</u> to take care of modeling any important phenomena that are unaccounted for by the generic objects.

## <u>2.3.1. The basic objects</u>

We begin by defining one ROSS generic object for each of the kinds of real-world objects mentioned in the previous section. We call these objects <u>basic objects</u>. Each of these has several different attributes representing the structural knowledge associated with that type of object. For example, to express our structural knowledge of penetrators in ROSS, we create a generic object called PENETRATOR and define its attributes using the following ROSS command:

```
(ask MOVING-OBJECT create generic PENETRATOR with
     position            'a position'
     max-speed           'a maximum speed'
     speed               'current speed'
     bombs               'current number of bombs'
     status              'a status'
     flight-plan         'a flight plan')
```

where the phrases in single-quotes represent variables.

To capture the behaviors of each kind of real-world object, we begin by asking what different kinds of input messages each of these real-world objects could receive. For example, a fighter can receive a message (a) from its fighter base telling it to chase a penetrator under guidance from a radar, (b) from that radar telling it to vector to a projected intercept point with the penetrator, or (c) an 'in-range' message informing it that the penetrator is in its radar range. Each of these messages then becomes the basis for a fighter behavior written in ROSS. To determine the structure of each of these behaviors, we ask what messages the object transmits in response to each of its inputs. For example, in response to a 'chase' message from its fighter base, a fighter will send a message to itself to take off, and then send a message to the specified radar requesting guidance to the penetrator. The following ROSS command captures this behavior:

```
(ask FIGHTER when receiving (chase >penetrator guided by >GCI)
  (~you unplan all (land))
  (~you set your status to scrambled)
  (if (~you are on the ground) then
      (~you take off))
  (~requiring (~your guide-time) tell ~the GCI guide ~yourself
                 to ~the penetrator)).
```

(The '~'s signal abbreviations. The ROSS abbreviations package [MCA82] enables the user to introduce English-like expressions into his programs and to tailor the expressions to his own preference. This approach towards readability is particularly flexible, since the user is not restricted to any system-defined English interface.)

### 2.3.2. Organizing the basic objects into a hierarchy

The behaviors of basic objects often have many commonalities that are revealed in the process of defining their behaviors. For example, GCIs, AWACS and SAMs all share the ability to detect, and their detection behaviors are identical. We can take advantage of ROSS's inheritance hierarchy (see [MCA82]) to reorganize object behaviors in a way that both emphasizes these conceptual similarities and eliminates redundant code. For example, for objects that have the behaviors of detection in common, we define a more abstract generic object called RADAR, to store these common behaviors. We then place it above GCI, AWACS and SAM in the hierarchy, so that they automatically inherit the behavior for detection whenever necessary. Hence we avoid writing these behaviors separately three times.

By applying this logic to other sets of basic objects and their common behaviors, we arrived at the following hierarchical organization of basic objects or object types in SWIRL:

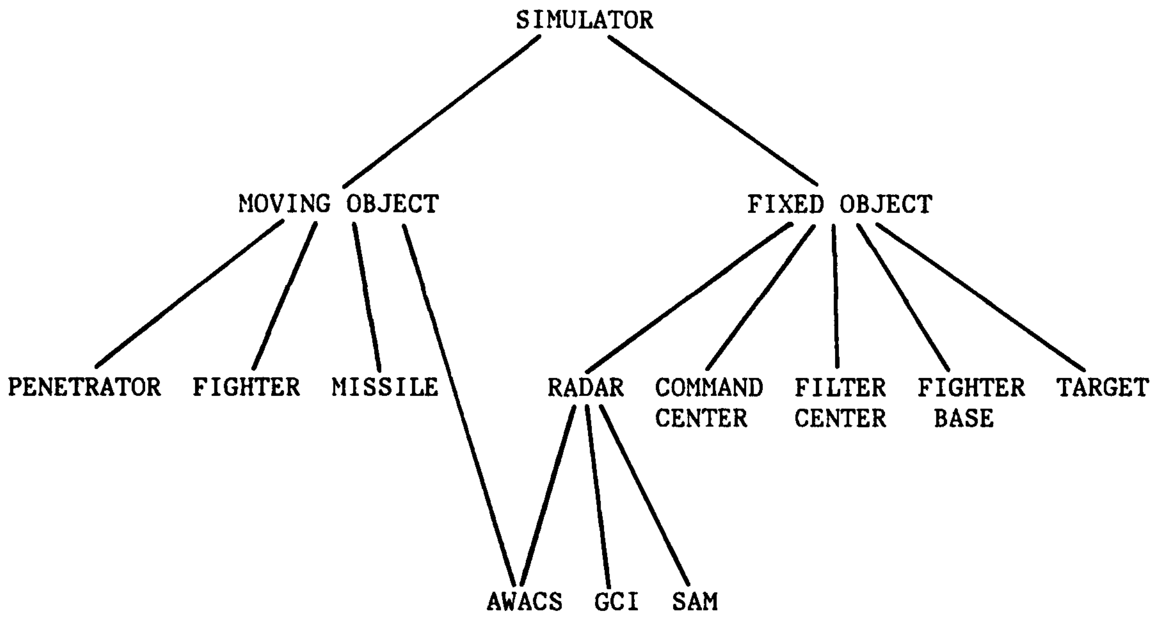


Figure 2. SWIRL Hierarchy of Basic Objects

Each object type in the class hierarchy can be construed as a description or view of the objects below it. One object, AWACS, happens to inherit its attributes and behaviors along more than one branch of the hierarchy (via RADAR and MOVING-OBJECT). Such 'multiple views' or 'multiple-inheritance' is possible in ROSS but not in most other object-oriented programming environments.

### 2.3.3. Modeling non-intentional events

The basic objects and their behaviors have a clear correspondence to real-world objects and their responses to the deliberate actions of others. These actions comprise most of the significant events that we wish to simulate. However, there are several important kinds of events that represent side effects of deliberate actions (e.g., a penetrator appearing as a blip on a radar screen is a side effect of the penetrator flying his course and entering a radar range). Such events are important since they may trigger other actions (e.g., a radar detecting a penetrator and notifying a filter center). However, these non-intentional events do not correspond to real-world message transmissions (e.g., a penetrator does not notify a radar that it has entered the radar's range). An important issue in the development of SWIRL has been how to capture these non-intentional events in an object-oriented framework (i.e., via message transmissions).

One method of capturing non-intentional events could be to refine the grain of simulation. The grain of a simulation is determined by the kind of real-world object one chooses to represent as a ROSS object. A division of the air-battle domain into objects like penetrators and radars is relatively coarse grained; a finer grain is possible. In particular, one could choose to create objects that represent small parts of the airspace through which penetrators fly. Then, as penetrators move they would send messages to those sectors that they were entering or leaving (just as objects moving through real space impact or 'send messages' to that space). Sectors could be associated with radars whose ranges they define, and act as intermediary objects to notify radars when penetrators enter their ranges. Essentially this solution proposes modeling the situation at an almost 'molecule-strikes-molecule' level of detail since, by adopting this level, one can achieve a strict mechanical cause-and-effect chain that is simple to model via message transmissions between real objects.

However, although this method solves one modeling problem, it causes two others that make it intractable. First, the method entails a prohibitive amount of computation. Second, in most cases, the extra detail would make modeling very awkward and unnatural (at least for our purposes in building and using SWIRL). The natural level of decomposition is that of 'coarse objects' such as penetrator and fighter. To the extent we stray from this, we make the simulation writer's job more difficult since he can no longer conceive of the task in the way that comes simplest or most naturally to him. In summary we reject this technique because it violates the following principle, which we have found useful in designing object-oriented simulators:

> THE APPROPRIATE DECOMPOSITION PRINCIPLE: Select a level of decomposition into objects that is 'natural' and at a level of detail commensurate with the goals and purposes of the model.

A second solution for modeling non-intentional events would be to allow the basic objects themselves to transmit the appropriate messages.

For example, if we allow a penetrator (with a fixed route) to see the position and ranges of all radars, it could compute when it would enter those ranges and send the appropriate 'in-range' messages to the radars. This solution is computationally tractable. However it has the important drawback that it allows the penetrator to access pieces of knowledge that, in reality, it cannot access. Penetrators in the real world know their routes but they may not know the location of all enemy radars. Even if they did, they do not send messages to radars telling the radars about themselves. In short, we reject this technique because it violates another useful principle that can be formulated as follows:

> THE APPROPRIATE KNOWLEDGE PRINCIPLE: Try to embed in your objects only legitimate knowledge, i.e., knowledge that can be directly accessed by the real-world objects that are being modeled.

### 2.3.4. Auxiliary objects

We feel that the above principles should be considered by anyone attempting to develop an object-oriented simulation, as they are critical to insure readable and conceptually clear code. The principles represent fairly severe constraints on programming style in object-oriented simulation. However, they do not over-constrain. The solution we offer in SWIRL represents one technique that adheres to both principles.

We introduce the notion of an auxiliary object. An auxiliary object is a full object in the ROSS sense. However, it does not have a real-world correlate. Nevertheless, it turns out to be a useful device for handling certain computations that cannot be naturally delegated to real objects. We have included three auxiliary objects in SWIRL:

1. The Scheduler. The scheduler may be interpreted as an omniscient, god-like being which, given current information, anticipates non-intentional events in the future and informs the appropriate objects as to their occurrence.

2. The Physicist. This object accounts for the effects of physical phenomena such as bomb explosions and ecm (electronic counter measures).

3. The Mathematician. This object executes all complex mathematical computations. With its help, it is possible to remove mathematical details from the top-level behaviors of objects, producing particularly readable code, e.g.,

```
    (ask mathematician determine time and position of
       interception of ~the fighter with ~the penetrator).
```

The behaviors of the auxiliary and basic objects are documented in Section 5, and the code for each behavior can be found in Section 6.

## 2.4. Operation

At the top level of the simulation, each of the penetrators and AWACS is asked to commence flight. For ground radars (e.g., GCIs, SAMs), potential interactions, resulting from penetrators entering their radar range, are computed and scheduled. Airborne radars (e.g., AWACS) are told to begin looking for incoming penetrators. (Note that moving radars will need to check for penetrators more often that ground radars because of their constantly changing positions.) Thus, for efficiency, SWIRL tries to anticipate when penetrators will fly into the range of any radars and to schedule appropriate transmissions of 'in-range' messages to these radars.

After this initialization, the clock begins to advance in time steps corresponding to a user-defined "ticksize." At the appropriate time, scheduled 'in-range' messages are sent to radars that, in turn, respond by sending messages to other objects. 'In-range' messages thus trigger chains of message transmissions. In particular, objects could themselves schedule messages for transmission in the future. The entire simulation proceeds in this manner, through a sequence of cycles of message-scheduling, time-advancement, message-transmissions.

SWIRL is predominantly an event-based simulator. Events correspond to messages. Although time is advanced in specified increments, only those events within the current time segment are processed. The time steps are needed for graphics processing. When graphics is operational, each moving object must update its position at each time step. This is to give the viewer a realistic animated simulation. However, interactions between objects continue to be event-based.

## 3.0. COMPUTING ENVIRONMENT

The ROSS interpreter is written in MACLISP and runs under the TOPS-20 operating system on a DEC20 (KL10). Space requirements for this interpreter are about 112K 36 bit words. SWIRL currently contains the fourteen generic objects shown in Figure 2 plus the three auxiliary objects mentioned above, along with approximately 175 behaviors. Compiled SWIRL code uses about 48K words. The largest SWIRL simulation environment contains well over 100 individual objects and the file defining these objects uses about 3K words. Total CPU usage for the simulation of an air-battle about three hours long is about 95 seconds. This includes time for the computation needed to drive the graphics interface.

We use two graphics processors, an AED 512 and a Genisco 3000. They are currently both driven by a PDP 11/45 which is connected to the DEC20 via a 9600 Baud link. The graphics software is implemented in C under the UNIX operating system.

# 4.0. USER INTERACTION WITH SWIRL

There are several levels on which the user can interact with SWIRL. Each succeeding level increases the degree to which the user modifies SWIRL and tailors it to his own conventions. Additionally, each level presumes an increasing familiarity with the underlying SWIRL implementation and with the ROSS language.

## 4.1. Running an existing SWIRL simulation environment

SWIRL code may be considered as consisting of two parts. One part contains all the generic objects (basic and auxiliary objects) and all of their behaviors. The second part contains individual instances of the generic objects, e.g., the particular offensive and defensive forces and their associated data (locations, velocity, weapon loads, etc.). We call this latter part the "simulation environment."

At the simplest level the user may run SWIRL using a pre-defined simulation environment (one that we have formulated). Such a user may be interested in seeing an example of SWIRL in action, or gathering data about a specific existing simulation. To run an existing SWIRL simulation, the user may follow these steps:

1. Get into ROSS by typing 'ROSS' at the TOPS-20 monitor-level.
2. Once in ROSS type '(LLOAD |<ROSS.SWIRL>SWIRL.LSP|)' to load the SWIRL user interface. (The interface file, SWIRL.LSP, is in the directory <ROSS.SWIRL>. Users may access files in this directory, e.g., to load or execute them or to copy them into their own directory. However, users will not be able to make changes to <ROSS.SWIRL> files.)
3. When you see the menu, load either the interpreted or compiled SWIRL code (select menu option 3 or 2 respectively).

4. Now load a specific simulation environment by selecting option 5.

5. When prompted for the name of the simulation file containing that environment, type it.

6. If you wish to record any message transmissions during a run, e.g., for event tracing, select menu option 8, then answer all following prompts.

7. Finally select option 6 or 7 to run the simulation. Option 6 enables one to watch a graphical rendition of the simulation, while option 7 runs the simulation without graphics.

These steps are exemplified in the following sample user session:

```
@ROSS

                    ROSS VERSION:  December 5. 1981.

See ross.news10 for recent changes

;Loading LOOP 725NIL
(LLOAD |<ROSS.SWIRL>SWIRL.LSP|)

Select option:
 1 -- Break into ROSS
 2 -- Load compiled SWIRL
 3 -- Load interpreted SWIRL
 4 -- Recompile interpreted SWIRL files
 5 -- Load a simulation environment
 6 -- Run simulation with graphics
 7 -- Run simulation without graphics
 8 -- Activate historian and reporter
 9 -- Browse or edit behaviors
10 -- Exit SWIRL & ROSS    3

Loading interpreted SWIRL
[LLOAD of file PS:<ROSS.SWIRL>FNS.LSP.2 completed.]

Select option:
 1 -- Break into ROSS
 2 -- Load compiled SWIRL
 3 -- Load interpreted SWIRL
 4 -- Recompile interpreted SWIRL files
 5 -- Load a simulation environment
 6 -- Run simulation with graphics
 7 -- Run simulation without graphics
 8 -- Activate historian and reporter
 9 -- Browse or edit behaviors
10 -- Exit SWIRL & ROSS    5

Source for simulation:  SWIRL-ALL
```

<ROSS assumes the file SWIRL-ALL.LSP will be in the user's directory. To use an existing SWIRL environment, the user may copy one from the directory <ROSS.SWIRL> to his own directory, or, alternatively, use an existing one in the <ROSS.SWIRL> directory; e.g., by typing |<ROSS.SWIRL>SWIRL-ALL| above.>

```
SWIRL loading completed

Select option:
 1 -- Break into ROSS
 2 -- Load compiled SWIRL
 3 -- Load interpreted SWIRL
 4 -- Recompile interpreted SWIRL files
 5 -- Load a simulation environment
 6 -- Run simulation with graphics
 7 -- Run simulation without graphics
 8 -- Activate historian and reporter
 9 -- Browse or edit behaviors
10 -- Exit SWIRL & ROSS    8

Specify file to record history:  RECORD
```

<Creates the output file RECORD.LSP in the user's directory.>

```
Output to terminal (T or NIL):  T

    1. -- PENETRATOR
    2. -- FIGHTER
    3. -- GCI
    4. -- SAM
    5. -- AWACS
    6. -- RADAR
    7. -- FILTER-CENTER
    8. -- COMMAND-CENTER
    9. -- FIGHTER-BASE
    10. -- TARGET
    11. -- MISSILE
    12. -- MOVING-OBJECT
    13. -- FIXED-OBJECT
    14. -- SCHEDULER
    15. -- PHYSICIST
    16. -- MATHEMATICIAN
Give list of objects in parentheses, e.g., (1 4 8), or NIL:  (1 2 4 5 6 7 9)

PENETRATOR has the following message templates:
    1. -- (FLY TO >PLACE)
    2. -- (DROP >M MEGATON BOMBS EXPLODING AT ALTITUDE >H)
    3. -- (>RADAR IS NOW TRACKING YOU)
    4. -- (>RADAR IS NO LONGER TRACKING YOU)
    5. -- (EVADE)
    6. -- (RESCHEDULE YOUR NEXT SECTOR)
    7. -- (MAKE A RANDOM TURN)
    8. -- (MAKE A TURN >N DEGREES >DIRECTION)
```

```
Give list of message numbers to record (T for all):  T

FIGHTER has the following message templates:
    1. -- (CHASE >PENETRATOR GUIDED BY >GCI)
    2. -- (ARE ON THE GROUND)
    3. -- (TAKE OFF)
    4. -- (CHASE >PENETRATOR TO >POSITION)
    5. -- (FOLLOW UNGUIDED POLICY WITH >PENETRATOR)
    6. -- (ENGAGE >PENETRATOR)
    7. -- (>PENETRATOR IS IN YOUR RANGE)
    8. -- (COMMENCE END GAME WITH >PENETRATOR)
    9. -- (COMPUTE THE END GAME RESULT)
    10. -- (DECREMENT YOUR MISSILES)
    11. -- (RETURN TO BASE)
    12. -- (LAND)
    13. -- (REARM)
    14. -- (LOOK FOR >OBJECT)
    15. -- (STOP LOOKING FOR >PENETRATOR)
    16. -- (>PEN IS OUT OF YOUR RANGE)
    17. -- (HAS ENOUGH MISSILES)
Give list of message numbers to record (T for all):  T

SAM has the following message templates:
    1. -- (EXPECT >PENETRATOR)
    2. -- (>PEN HAS CHANGED ROUTE)
    3. -- (FIRE AT >PENETRATOR)
    4. -- (>MISSILE DESTROYED >PENETRATOR)
    5. -- (>MISSILE MISSED >PENETRATOR)
Give list of message numbers to record (T for all):  T

AWACS has the following message templates:
    1. -- (LOOK FOR >OBJECT)
Give list of message numbers to record (T for all):  T

RADAR has the following message templates:
    1. -- (>THING IS IN YOUR RANGE)
    2. -- (TRANSMIT TO YOUR FILTER-CENTER THAT +MESSAGE)
    3. -- (>PENETRATOR IS OUT OF YOUR RANGE)
    4. -- (TRY TO CHANGE GUIDER OF >FIGHTER TO >PENETRATOR)
    5. -- (>PENETRATOR IS DESTROYED)
    6. -- (>PENETRATOR HAS CHANGED ROUTE)
    7. -- (IS >PENETRATOR STILL IN YOUR RANGE)
    8. -- (GUIDE >FIGHTER TO >PENETRATOR)
    9. -- (STOP GUIDING >FIGHTER)
    10. -- (FIND A NEW GCI TO GUIDE >FIGHTER TO >PENETRATOR)
    11. -- (>FIGHTER HAS SIGHTED >PENETRATOR)
    12. -- (>FIGHTER UNABLE TO CHASE >PENETRATOR)
    13. -- (ARE SATURATED)
    14. -- (ARE NOT ECMED OUT BY >PENETRATOR)
    15. -- (CHECK FOR NEW PENETRATORS)
Give list of message numbers to record (T for all):  T

FILTER-CENTER has the following message templates:
    1. -- (>THING IS IN RANGE OF >GCI)
    2. -- (PREPARE TO DEFEND AGAINST >THING MONITORED BY >GCI)
    3. -- (DETERMINE >THING IS A HOSTILE PENETRATOR)
    4. -- (ALERT YOUR FIGHTER-BASES ABOUT >PENETRATOR)
    5. -- (ALERT YOUR SAMS ABOUT >PENETRATOR)
    6. -- (TRANSMIT TO >AGENT TO +DIRECTIVE)
    7. -- (TRANSMIT TO YOUR COMMAND-CENTER THAT +MESSAGE)
    8. -- (>GCI HAS LOST >PENETRATOR)
    9. -- (>PENETRATOR KILLED BY >AGENT)
    10. -- (>AGENT MISSED >PENETRATOR)
Give list of message numbers to record (T for all):  T

FIGHTER-BASE has the following message templates:
    1. -- (ACTIVATE)
    2. -- (WIND DOWN)
    3. -- (SCRAMBLE SOME FIGHTERS GUIDED BY >GCI TO >PENETRATOR)
    4. -- (DETERMINE WHICH OF >FIGHTERS IS NEAREST >PENETRATOR)
    5. -- (SEND >FIGHTER GUIDED BY >GCI TO >PENETRATOR)
    6. -- (>FIGHTER UNABLE TO CHASE >PENETRATOR)
    7. -- (PUT >FIGHTER ON YOUR LIST OF >ATTRIB)
Give list of message numbers to record (T for all):  T

Select option:
 1 -- Break into ROSS
 2 -- Load compiled SWIRL
 3 -- Load interpreted SWIRL
 4 -- Recompile interpreted SWIRL files
 5 -- Load a simulation environment
 6 -- Run simulation with graphics
 7 -- Run simulation without graphics
 8 -- Activate historian and reporter
 9 -- Browse or edit behaviors
10 -- Exit SWIRL & ROSS      7
```

<Note that option 6, which includes graphics output, is operational only at Rand.>

```
Type number of ticks to run:   100
```

<Simulation begins and its verbal output is shown below. Numbers on the left give the clock time at which the specified event occurred. The event (which is also a message) is represented by a list. The first member of the list specifies the object that received the message contained in the rest of the list. For example, at time 492.891342, AWACS1 received the message (PEN2 IS IN YOUR RANGE).>

```
               .
               .
0.0   (AWACS3 LOOK FOR PEN3)
0.0   (AWACS3 LOOK FOR PEN2)
0.0   (AWACS3 LOOK FOR PEN1)
0.0   (AWACS2 LOOK FOR PEN3)
0.0   (AWACS2 LOOK FOR PEN2)
0.0   (AWACS2 LOOK FOR PEN1)
0.0   (AWACS1 LOOK FOR PEN3)
0.0   (AWACS1 LOOK FOR PEN2)
```

```
0.0   (AWACS1 LOOK FOR PEN1)
0.0   (PEN3 FLY TO (880.0 210.0))
0.0   (PEN2 FLY TO (352.0 700.0))
  0.0   (GCI3 IS PEN2 STILL IN YOUR RANGE)
  0.0   (GCI2 IS PEN2 STILL IN YOUR RANGE)
0.0   (PEN1 FLY TO (220.0 1085.0))
492.891342   (AWACS1 PEN2 IS IN YOUR RANGE)
  492.891342   (AWACS1 ARE NOT ECMED OUT BY PEN2)
934.89134   (AWACS1 PEN2 IS OUT OF YOUR RANGE)
1817.13728   (GCI2 PEN2 IS IN YOUR RANGE)
  1817.13728   (GCI2 ARE NOT ECMED OUT BY PEN2)
1817.13728   (PEN2 GCI2 IS NOW TRACKING YOU)
1848.00002   (PEN1 FLY TO (660.0 840.0))
  1848.00002   (GCI1 IS PEN1 STILL IN YOUR RANGE)
1967.13742   (GCI3 PEN2 IS IN YOUR RANGE)
  1967.13742   (GCI3 ARE NOT ECMED OUT BY PEN2)
  1967.13742   (GCI3 ARE SATURATED)
  1967.13742   (GCI3 TRANSMIT TO YOUR FILTER-CENTER THAT
                               PEN2 IS IN RANGE OF GCI3)
1967.13742   (PEN2 GCI3 IS NOW TRACKING YOU)
2027.13742   (FC1 PEN2 IS IN RANGE OF GCI3)
  2027.13742   (FC1 DETERMINE PEN2 IS A HOSTILE PENETRATOR)
2117.13742   (FC1 PREPARE TO DEFEND AGAINST PEN2 MONITORED BY GCI3)
  2117.13742   (GCI3 IS PEN2 STILL IN YOUR RANGE)
  2117.13742   (FC1 ALERT YOUR FIGHTER-BASES ABOUT PEN2)
    2117.13742   (FC1 TRANSMIT TO FTB1 TO ACTIVATE)
    2117.13742   (FC1 TRANSMIT TO FTB2 TO ACTIVATE)
    2117.13742   (FC1 TRANSMIT TO FTB3 TO ACTIVATE)
  2117.13742   (FC1 ALERT YOUR SAMS ABOUT PEN2)
  2117.13742   (FC1 TRANSMIT TO YOUR COMMAND-CENTER THAT
                         PEN2 MONITORED BY GCI3 IS HOSTILE)
2177.13742   (FTB1 ACTIVATE)
2177.13742   (FTB2 ACTIVATE)
2177.13742   (FTB3 ACTIVATE)
2197.13742   (FTB2 SCRAMBLE SOME FIGHTERS GUIDED BY GCI3 TO PEN2)
  2197.13742   (FTB2 DETERMINE WHICH OF (FT11 FT12 FT13 FT14 FT15
                         FT16 FT17 FT18 FT19 FT20) IS NEAREST PEN2)
  2197.13742   (FT11 HAS ENOUGH MISSILES)
  2197.13742   (FTB2 SEND FT11 GUIDED BY GCI3 TO PEN2)
2207.13742   (FT11 CHASE PEN2 GUIDED BY GCI3)
  2207.13742   (FT11 ARE ON THE GROUND)
  2207.13742   (FT11 TAKE OFF)
2247.13742   (GCI3 GUIDE FT11 TO PEN2)
2277.13742   (FT11 CHASE PEN2 TO (379.976994 700.0))
2359.83917   (AWACS3 PEN3 IS IN YOUR RANGE)
  2359.83917   (AWACS3 ARE NOT ECMED OUT BY PEN3)
  2359.83917   (AWACS3 ARE SATURATED)
  2359.83917   (AWACS3 TRANSMIT TO YOUR FILTER-CENTER THAT
                                 PEN3 IS IN RANGE OF AWACS3)
2376.0   (PEN2 FLY TO (660.0 490.0))
  2376.0   (GCI5 IS PEN2 STILL IN YOUR RANGE)
  2376.0   (GCI4 IS PEN2 STILL IN YOUR RANGE)
  2376.0   (GCI3 PEN2 HAS CHANGED ROUTE)
    2376.0   (GCI3 GUIDE FT11 TO PEN2)
  2376.0    (GCI2 PEN2 HAS CHANGED ROUTE)
2406.0    (FT11 CHASE PEN2 TO (389.13874 674.67813))
2419.83917    (FC2 PEN3 IS IN RANGE OF AWACS3)
  2419.83917    (FC2 DETERMINE PEN3 IS A HOSTILE PENETRATOR)
2509.83917    (FC2 PREPARE TO DEFEND AGAINST PEN3 MONITORED BY AWACS3)
  2509.83917    (AWACS3 IS PEN3 STILL IN YOUR RANGE)
  2509.83917    (FC2 ALERT YOUR FIGHTER-BASES ABOUT PEN3)
    2509.83917    (FC2 TRANSMIT TO FTB4 TO ACTIVATE)
    2509.83917    (FC2 TRANSMIT TO FTB5 TO ACTIVATE)
  2509.83917    (FC2 ALERT YOUR SAMS ABOUT PEN3)
  2509.83917    (FC2 TRANSMIT TO YOUR COMMAND-CENTER THAT
                              PEN3 MONITORED BY AWACS3 IS HOSTILE)
2519.02338    (GCI2 PEN2 IS OUT OF YOUR RANGE)
2519.02338    (PEN2 GCI2 IS NO LONGER TRACKING YOU)
2569.83917    (FTB4 ACTIVATE)
2569.83917    (FTB5 ACTIVATE)
2589.83917    (FTB4 SCRAMBLE SOME FIGHTERS GUIDED BY AWACS3 TO PEN3)
  2589.83917    (FTB4 DETERMINE WHICH OF (FT31 FT32 FT33 FT34 FT35 FT36
                                  FT37 FT38 FT39 FT40) IS NEAREST PEN3)
  2589.83917    (FT31 HAS ENOUGH MISSILES)
  2589.83917    (FTB4 SEND FT31 GUIDED BY AWACS3 TO PEN3)
2599.83917    (FT31 CHASE PEN3 GUIDED BY AWACS3)
  2599.83917    (FT31 ARE ON THE GROUND)
  2599.83917    (FT31 TAKE OFF)
2639.0548    (GCI4 PEN2 IS IN YOUR RANGE)
  2639.0548    (GCI4 ARE NOT ECMED OUT BY PEN2)
2639.0548    (PEN2 GCI4 IS NOW TRACKING YOU)
2639.83917    (AWACS3 GUIDE FT31 TO PEN3)
2645.6987    (FT11 ENGAGE PEN2)
  2645.6987    (FT11 RETURN TO BASE)
    2645.6987    (FTB2 PUT FT11 ON YOUR LIST OF FIGHTERS-AVAILABLE)
2669.83917    (FT31 CHASE PEN3 TO (857.64428 174.434078))

            .
            .
<Simulation continues for 100 ticks>
            .
            .

Select option:
 1 -- Break into ROSS
 2 -- Load compiled SWIRL
 3 -- Load interpreted SWIRL
 4 -- Recompile interpreted SWIRL files
 5 -- Load a simulation environment
 6 -- Run simulation with graphics
 7 -- Run simulation without graphics
 8 -- Activate historian and reporter
 9 -- Browse or edit behaviors
10 -- Exit SWIRL & ROSS      10

Goodbye.

@    <This puts you into TOPS-20.  Type CONTINUE to get menu back.>
```

## 4.2. Defining a new SWIRL simulation environment

The next level of interaction with SWIRL enables the user to define a new simulation environment, while keeping the SWIRL behaviors constant. To create a new environment, one needs to create specific instances of the generic objects already defined by SWIRL. SWIRL, by itself, creates no specific object instances. This is left entirely up to the user.

The primary method for defining a new simulation environment is:

1. Create a file with a .LSP extension (e.g., SWDEMO.LSP).
2. Edit the file (using EMACS or some other editor), inserting into it ROSS commands that create instances of the generic objects, and giving each instance specific values for the properties required by that type of object. (If EMACS is to be used, the file <maclisp>emacs.init must be copied into the user's directory.)
3. Enter ROSS, load the SWIRL code, and select menu option 5.
4. Respond to the prompt with the name of the .LSP file (do not actually type the .LSP extension; thus, for example, type SWDEMO)

Steps 3 and 4 of this procedure have already been illustrated above. Steps 1 and 2 are exemplified below:

```
@EMACS SWDEMO.LSP

  <Enter EMACS, our favorite editor [STA81]>

; -------------- initialize GCIs ---------------------------------

(tell GCI create instance GCI1  with
      position (425.0 900.0)
      filter-center FC1
      Range 100.0
      Status Active)

(tell GCI create instance GCI2  with
      position (325.0 775.0)
      filter-center FC1
      Range 100.0
      Status Active)

; ------------ initialize filter centers --------------------------

(tell filter-center create instance FC1 with
      position (506.0  700.0)
      command-center cc1
      Status Active)

(tell filter-center create instance FC2 with
      position (616.0  385.0)
      command-center cc1
      Status Active)

; ------------ initialize fighter bases ---------------------------

(tell fighter-base create instance FTB1 with
      position (600.0  850.0)
      filter-center FC1
      Status active
      fighters-available (FT1 FT2 FT3 FT4 FT5 FT6 FT7 FT8 FT9 FT10)
      fighters-scrambled nil
      Range 400.0)

; ------------ initialize fighters --------------------------------

(tell fighter create instance Ft1 with base FTb1 Missiles 6 Fuel 6000.
                                         Status Active)
(tell fighter create instance Ft2 with base FTb1 Missiles 6 Fuel 6000.
                                         Status Active)
(tell fighter create instance Ft3 with base FTb1 Missiles 6 Fuel 6000.
                                         Status Active)
```

<etc.; then save the file.>

Because the specification of generic objects and their behaviors are kept completely separate from specific simulation environments, the user will find it simple to create many diverse air-battle simulations without knowledge of the SWIRL code for the various behaviors. For example, by merely changing 'position' attributes one can investigate different designs for radar-placement patterns, or assess the value of a 'two-wave' versus a 'single-wave' offensive attack. To add even greater flexibility to simulation environment specification, the user is not restricted to defining simulation environments in loadable files. It is always possible to issue commands, like those directly above, when interacting with ROSS. However, it is usually more efficient to put large simulation definitions that will be repeatedly accessed into a file, and leave only minor tailoring for interactive specification.

## 4.3. Changing SWIRL behaviors

To investigate some air-battle designs, a user may need still more flexible ways of modifying SWIRL than those afforded by the techniques discussed above. The most powerful method of modification is to change SWIRL behaviors. This can be accomplished by editing a behavior file (modifying an existing file or creating a new one) and then loading it within the ROSS/SWIRL environment. For small changes however, the user may choose to remain in ROSS and follow these steps:

1. Select option 1 on the ROSS menu.
2. Type in new behavior definitions (or, alternatively, load a file containing new behaviors).
3. Hit CTRL/K to get the SWIRL menu back.

In order to change behaviors, the user needs to be familiar with the SWIRL code. However, since the SWIRL code is very modular, this familiarization may be surprisingly modest. Typically, the user will just want to alter what a given object does in response to a small number of messages. For example, by default, the response of SWIRL's penetrators when receiving a 'radar is no longer tracking you' message (i.e., the penetrator detects that it has just left radar coverage) is to continue flying on a straignt course. This is a simple response, but perhaps not appropriate. The user might want to investigate the effects of a more complex strategy such as an evasive maneuver. This can be easily accomplished by redefining the behavior in question as follows:

```
    (ask penetrator when receiving (>radar is no longer detecting you)
         (~you plan after 30 seconds make a turn 90 degrees right)
         (~you reschedule your next sector))
```

(Note: A user intending a temporary change to the above behavior could just issue this in a ROSS session. If this is a permanent change then it should be put into a file.)

This example demonstrates that one reason substantial behavior modification can be done very easily is that SWIRL provides a large number of behavioral building blocks (here, the "(make a turn >n degrees >direction)" behavior). These represent behaviors endowed to various types of objects that may be used for several purposes but that are often not 'hooked into' obvious places. As another example, missile firings were first modeled probabilistically in SWIRL. However, one can also treat missiles as full simulation objects and integrate them into the simulation by simple changes to the behaviors of SAMs (that fire them). Our philosophy in building SWIRL was to keep many strategic and tactical behavior specifications simple and encourage the user to add realistic complexity to them as an exercise in design. The existence of behavioral building blocks often makes this design process as simple as judiciously 'composing' two or more existing behaviors.

Naturally, we may not have provided all the appropriate building blocks. In this case, the user will have to define his own from much simpler SWIRL behaviors and primitive ROSS commands. This requires a fairly intimate knowledge of ROSS and the existing code. The user intending these kinds of modifications is referred to Sections 5 and 6 of this paper, where the SWIRL code is presented and documented, and to the ROSS Language Manual [MCA82]. We hope, however, that the user will only infrequently need to descend into the 'nether regions' of SWIRL. Moreover, if the user follows our example in building generally useful behavioral building blocks, he should soon have a fairly complete repertoire of domain-specific primitives. Then, most design changes may be accomplished in a very small number of moves.

When behaviors are changed, they may be recompiled using option 4 on the SWIRL menu. Currently, however, this option may be used only by the ROSS/SWIRL research group.

## 4.4. Reading SWIRL code and documentation

We have tried to make SWIRL code as English-like and as readable as possible. So the code itself could be considered its own documentation. In addition, however, English descriptions of all generic object behaviors have also been written. These are organized once again as a set of ROSS commands. Below we outline how the SWIRL code or the English documentation may be accessed.

After typing 1 at the menu, and breaking into ROSS, one can type "(ask browser help)" and get directions on how to browse through the SWIRL code. Alternatively, one may choose menu option 9 and may, for example, go through the following session:

```
Select option:
 1 -- Break into ROSS
 2 -- Load compiled SWIRL
 3 -- Load interpreted SWIRL
 4 -- Recompile interpreted SWIRL files
 5 -- Load a simulation environment
 6 -- Run simulation with graphics
 7 -- Run simulation without graphics
 8 -- Activate historian and reporter
 9 -- Browse or edit behaviors
10 -- Exit SWIRL & ROSS   9
```

<Option 9 allows the ROSS/SWIRL research group to modify SWIRL code and documentation; however, it will not allow users to do so. Users should not attempt to modify and save files edited via option 9. Note that when using option 9, the EMACS editor will be invoked. It assumes that the file emacs.init exists in the user's directory.>

```
Recording documentation information ...

DOC-FILE= PS:<ROSS.SWIRL>AWACS.DOC.0
DOC-FILE= PS:<ROSS.SWIRL>BACON.DOC.0
         .
         .

DOC-FILE= PS:<ROSS.SWIRL>SWIRL.DOC.0
```

<For each generic object, the file containing documentation on it is loaded into main memory. Documentation files have a .DOC extension.>

<Alternatively type "(ask browser help)" when outside menu.>

```
    1. -- PENETRATOR
    2. -- FIGHTER
    3. -- GCI
    4. -- SAM
    5. -- AWACS
    6. -- RADAR
    7. -- FILTER-CENTER
    8. -- COMMAND-CENTER
    9. -- FIGHTER-BASE
    10. -- TARGET
    11. -- MISSILE
    12. -- MOVING-OBJECT
    13. -- FIXED-OBJECT
    14. -- SCHEDULER
    15. -- PHYSICIST
    16. -- MATHEMATICIAN
Give number of object you wish to examine, or NIL to stop:   6

Documentation is available on the following templates:
    1. -- (>THING IS IN YOUR RANGE)
    2. -- (TRANSMIT TO YOUR FILTER-CENTER THAT +MESSAGE)
    3. -- (>PENETRATOR IS OUT OF YOUR RANGE)
    4. -- (TRY TO CHANGE GUIDER OF >FIGHTER TO >PENETRATOR)
    5. -- (>PENETRATOR IS DESTROYED)
    6. -- (>PENETRATOR HAS CHANGED ROUTE)
    7. -- (IS >PENETRATOR STILL IN YOJR RANGE)
    8. -- (GUIDE >FIGHTER TO >PENETRATOR)
    9. -- (STOP GUIDING >FIGHTER)
    10. -- (FIND A NEW GCI TO GUIDE >FIGHTER TO >PENETRATOR)
    11. -- (>FIGHTER HAS SIGHTED >PENETRATOR)
    12. -- (>FIGHTER UNABLE TO CHASE >PENETRATOR)
    13. -- (ARE SATURATED)
    14. -- (ARE NOT ECMED OUT BY >PENETRATOR)
    15. -- (CHECK FOR NEW PENETRATORS)
Give list of messages you wish to examine, or T for all,
or NIL to stop:   (8)

    1. -- ENGLISH DESCRIPTION
    2. -- SWIRL CODE
    3. -- BOTH
Type option, or NIL to stop:    1
```

<At this time, the file containing documentation for radar, <ROSS.SWIRL>RADAR.DOC, is "visited" (in the sense of EMACS), and the the cursor stops at the end of the template selected. The English description follows immediately. If the user elects to see the SWIRL code, the file <ROSS.SWIRL>RADAR.LSP will be visited.>

```
(ask radar documentation for
     (guide >fighter to >penetrator) is
|Sender: fighter or another radar handing over guidance of a fighter.
Guides a fighter to a pen. Four cases arise:
If the radar is destroyed it can do nothing. The fighter just follows
unguided policy.
If the fighter seeks guidance towards a penetrator that is not
currently tracked by the radar, it tries to find another radar to
guide the fighter. If this fails, it tells the fighter to follow
unguided policy.
If the radar is blinded, it tries to find another radar to guide the
fighter. If this fails, it tells the fighter to follow unguided
policy.
Otherwise the radar calculates an intercept point of the fighter with
the penetrator and tells the fighter to vector to this point.|)
```

<The user can read the documentation on this message template or the associated behavior of the radar (if he had chosen option 2 or 3 to the prompt above). Once within EMACS, the user will have full EMACS capabilities except saving the files. See [STA81] for EMACS operation description. To exit EMACS and return to the menu, the user should type CTRL/X followed by CTRL/Z.>

```
    1. -- PENETRATOR
    2. -- FIGHTER
    3. -- GCI
    4. -- SAM
    5. -- AWACS
    6. -- RADAR
    7. -- FILTER-CENTER
    8. -- COMMAND-CENTER
    9. -- FIGHTER-BASE
    10. -- TARGET
    11. -- MISSILE
    12. -- MOVING-OBJECT
    13. -- FIXED-OBJECT
    14. -- SCHEDULER
    15. -- PHYSICIST
    16. -- MATHEMATICIAN
Give number of object you wish to examine, or NIL to stop:   NIL

Select option:
 1 -- Break into ROSS
 2 -- Load compiled SWIRL
 3 -- Load interpreted SWIRL
 4 -- Recompile interpreted SWIRL files
 5 -- Load a simulation environment
 6 -- Run simulation with graphics
 7 -- Run simulation without graphics
 8 -- Activate historian and reporter
 9 -- Browse or edit behaviors
10 -- Exit SWIRL & ROSS   10

Goodbye.

@
  <This puts you into TOPS-20.  Type CONTINUE to get menu back.>
```

## 5.0. SWIRL DOCUMENTATION

This section presents detailed documentation for each behavior of each object in SWIRL. The objects that represent real-world entities will be presented first, followed by auxiliary objects. Documentation of each object begins with a short overview, followed by a description of each behavior. The descriptions are duplicated from output provided by the ROSS on-line browsing facility.

### 5.1. Penetrator

```
(comment                      Overview

|Penetrators have an initial flight plan, written in terms of commands like
(fly to <position>) and (drop <n> megaton bombs ...).  Their main behaviors
are concerned with executing these commands.  When  processing  a  (fly  to
<position>)  command, the penetrator determines all radar ranges the flight
segment might impact, and gives the scheduler enough  information  so  that
the  impacted  GCIs  will  be informed of the penetrator at the right time.
Penetrator also has a few behaviors for executing evasive turns.|)


(comment                     Behaviors                                   )

(tell penetrator documentation for
      (fly to >place) is
|Sender:  penetrator.  The segment between  penetrator's  current  position
and  'place'  represents  the  next  leg  of  flight.   Penetrator sets its
velocity to fly towards  'place'.   Penetrator  also  checks  if  it  would
interact  with any radars during this leg.  If it does it causes 'in-range'
messages  to  be  scheduled  for  the  radars  at  the  appropriate   time.
Scheduling in this fashion simulates tracking of penetrators.|)


(tell penetrator documentation for
      (drop >m megaton bombs exploding at altitude >h) is
|Sender:  penetrator.  Essentially tells  the  physicist  to  simulate  the
effects of explosion, and notifies the generic actor 'target'.|)


(tell penetrator documentation for
      (>radar is now tracking you) is
|Sender:  scheduler.  This message is sent to the penetrator  at  the  same
time  that the in-range message is sent to the 'radar'.  Currently there is
no action.|)
```

```
(tell penetrator documentation for
     (>radar is no longer tracking you) is
|Sender: scheduler. Penetrator upon receiving this message may choose
either to make an evasive turn or not.|)


(tell penetrator documentation for
     (evade) is
|Sender: penetrator. Penetrator makes a random turn either to the left or
to the right. After 'evade-duration' it resumes flying towards its
destination prior to making the turn.|)


(tell penetrator documentation for
     (reschedule your next sector) is
|Sender: penetrator. Penetrator resumes flying towards the point it was
before making the turn. |)


(tell penetrator documentation for
     (make a random turn) is
|Sender: penetrator. Penetrator makes a random turn 90 degrees left or
right.|)


(tell penetrator documentation for
     (make a turn >n degrees >direction) is
|Sender: penetrator. Penetrator sets its velocity to correspond to the
direction of turn ('n' degrees left or right).|)
```

## 5.2. Radar

```
(comment                    Overview

|Radar is a generic class whose subclasses include SAM, GCI and AWACS.
Radars have several basic functions. First they detect incoming
penetrators and relay this information up the chain of command and control.
Second, they guide fighters to intercept points with penetrators. Radars
may be blinded -- a side effect of explosions in the neighborhood--or
subject to electromagnetic counter measures. Either of these cause the
radar to be out of commission for guiding or detection.|)


(comment                    Behaviors                                )

(ask radar documentation for
    (>thing is in your range) is
|Sender: scheduler (which is generally responsible for determining
interactions of radar with static objects). Radar tells its filter center
after a reporting delay that an object has been detected. ECM and
saturation are taken into account.|)
```

(ask radar documentation for
(transmit to your filter-center that +message) is
|Sender: radar. This behavior implements message passing with a specific delay; in this case the reporting-delay.|)

(ask radar documentation for
(>penetrator is out of your range) is
|Sender: scheduler. Radar removes penetrator from its list and notifies its filter-center. If one of the objects not visible previously due to saturation, it becomes visible.|)

(ask radar documentation for
(try to change guider of >fighter to >penetrator) is
|Sender: the radar. Causes the radar to stop guiding the penetrator, find a new radar to guide the fighter. If successful, it unplans all its instructions to the fighter. If not successful, it instructs the fighter to follow unguided policy.|)

(ask radar documentation for
(find a new GCI to guide >fighter to >penetrator) is
|Sender: radar. When a penetrator goes out of radar's range the radar attempts to get another radar that is still tracking the penetrator to guide the first radar's 'fighter' to the penetrator.|)

(ask radar documentation for
(is >penetrator still in your range) is
|Sender: filter center. Returns true if the radar is still tracking the penetrator.|)

(ask radar documentation for
(guide >fighter to >penetrator) is
|Sender: fighter or another radar handing over guidance of a fighter. Guides a fighter to a pen. Four cases arise: If the radar is destroyed it can do nothing. The fighter just follows unguided policy. If the fighter seeks guidance towards a penetrator that is not currently tracked by the radar, it tries to find another radar to guide the fighter. If this fails, it tells the fighter to follow unguided policy. If the radar is blinded, it tries to find another radar to guide the fighter. If this fails, it tells the fighter to follow unguided policy. Otherwise the radar calculates an intercept point of the fighter with the penetrator and tells the fighter to vector to this point.|)

(ask radar documentation for
(stop guiding >fighter) is
|Sender: radar. Causes radar to remove the fighter from its list of objects guided and to unplan any future guiding behavior.|)

```
(ask radar documentation for
     (>fighter has sighted >penetrator)  is
|Sender:  fighter.  When a radar is told this, it stops guiding the fighter
and lets it go on its own.|)


(ask radar documentation for
     (>fighter unable to chase >penetrator)  is
|Sender:  fighter.  Radar unplans telling the  fighter  in  the  future  to
engage the 'penetrator'.|)


(ask radar documentation for
     (are saturated) is
|Sender:  radar.  Predicate returns 'true' if number of blips on screen  is
greater than the saturation limit.|)


(ask radar documentation for
     (are not ECMed out by >penetrator) is
|Sender:  radar.  Returns 'true' if radar can detect penetrator even though
the penetrator has turned on its ecm;  otherwise returns 'false'.|)


(ask radar documentation for
     (check for new penetrators) is
|Sender:  the radar.  If some penetrators were in the radar's range but the
radar  was  saturated,  this  makes  the  radar attend to the penetrator by
sending it an 'in-range' message.|)


(ask radar documentation for
     (>penetrator is destroyed) is
|Sender:  scheduler.  Radar sends this to all fighters chasing 'penetrator'
and  tells  them  to  return  to  base.  If radar was saturated, it can now
respond to one of the blips on its radar screen.|)


(ask radar documentation for
      (>penetrator has changed route) is
|Sender:   scheduler.   Radar   guides   afresh   all   fighters   chasing
'penetrator'.|)
```

## 5.3. GCI

```
(comment                      Overview

|Currently GCI inherits all its behaviors from radar.|)
```

## 5.4. AWACS

```
(comment                      Overview
```

|An AWACS inherits behaviors from both moving object and radar and (currently) has no unique behaviors of its own. It detects and guides like a radar (e.g., GCI), and it moves like any moving object.|)

(comment Behaviors )

(ask AWACS documentation for
    (look for >penetrator) is
|Sender: simulator initialization. Sets AWACS looking for penetrators.|)

## 5.5. SAM

(comment Overview

|SAMs have radars and fire missiles at penetrators. Missiles are treated as full ROSS objects. SAMs inherit all their detection behaviors from the radar object.|)

(comment Behaviors )

(tell SAM documentation for
    (expect >penetrator) is
|Sender: filter-center. SAM sets its status to 'alert'.|)

(tell SAM documentation for
    (>pen has changed route) is
|Sender: scheduler. Currently SAM takes no action.|)

(tell SAM documentation for
    (fire at >penetrator) is
|Sender: SAM to itself. Fires missiles at the 'penetrator'. The action is similar to that of fighter base and radar that respectively scramble and guide fighters to intercept a penetrator. SAM however does both, the launching and the guiding of the missile.|)

(tell SAM documentation for
    (>missile destroyed >penetrator) is
|Sender: 'Missile' that scored. SAM informs the scheduler that the 'penetrator' is destroyed.|)

(tell SAM documentation for
    (>missile missed >penetrator) is
|Sender: 'Missile' that missed. SAM checks if the 'penetrator' is still in its range and if so, fires another missile at it.|)

## 5.6. Missile

```
(comment                    Overview

|Missiles are shot by SAMs.  Functionally they are similar to fighters.|)


(comment                    Behaviors                              )

(tell missile documentation for
     (chase >penetrator to >position) is
|Sender:  SAM controlling missile.  The behavior is similar to the one  for
fighter.  Missile  just  vectors towards 'position' where it would hope to
hit 'penetrator'.|)


(tell missile documentation for
     (hit >penetrator)is
|Sender:  SAM controlling missile.  Missile attempts to hit penetrator.  It
either  succeeds  or  fails.   In  either case it reports the result to its
controlling SAM.|)
```

## 5.7. Filter Center

```
(comment                    Overview

|Filter centers receive initial radar reports from GCIs about  penetrators.
They  alert  all  their  known fighter bases and SAMs.  Finally they notify
their command center about the penetrator.|)


(comment                    Behaviors                              )

(ask filter-center documentation for
     (>thing is in range of >GCI) is
|Sender:  radar.  If the object is determined to  be  hostile,  the  filter
center prepares to defend against it.|)


(ask filter-center documentation for
     (determine >thing is a hostile penetrator) is
|Sender:  filter-center.  Should return 'true'  if  'thing'  is  a  hostile
penetrator.  Currently always 'true'.|)


(ask filter-center documentation for
     (prepare to defend against >thing monitored by >GCI) is
|Sender:  filter-center.  Alerts its fighter bases and SAMs,  and  notifies
its comand-center.|)


(ask filter-center documentation for
     (alert your fighter-bases about >penetrator) is
|Sender:   filter-center.   Alerts   all  associated  fighter-bases  in  the
penetrator's line of flight.|)


(ask filter-center documentation for
     (alert your SAMs about >penetrator) is
|Sender:  Filter-center.  Alerts all associated SAMs  in  the  penetrator's
line of flight.|)


(ask filter-center documentation for
     (transmit to >agent to +directive) is
|Sender:  filter-center.  This  behavior  essentially  expands  into  'plan
after' forms.  It is useful modeling communication delays.|)


(ask filter-center documentation for
     (transmit to your command-center that +message) is
|Sender:   filter-center.   After  the  appropriate  communication   delay,
'message' is transmitted to the command-center of the filter-center.|)


(ask filter-center documentation for
     (>GCI has lost >penetrator) is
|Sender:  radar.  If radar informs  filter-center  that  it  has  lost  the
penetrator,   the  filter-center  forgets  the  penetrator  and  tells  its
command-center.|)


(ask filter-center documentation for
     (>penetrator killed by >agent) is
|Sender:  SAM or fighter.  Message given when agent is successful;  used to
update filter-center's list of penetrators.|)


(ask filter-center documentation for
     (>agent missed >penetrator) is
|Sender:  SAM or fighter.  Message given when either associated fighter  or
SAM  misses  it's  assigned  penetrator.  Currently not used.  This message
could be the basis for some intelligent remedial action.|)
```

## 5.8. Fighter Base

```
(comment                     Overview

|Fighter bases receive messages to go on alert from filter centers whenever
new penetrators are discovered.  Command centers will request fighter bases
to scramble fighters to chase penetrators.  A GCI will guide the fighter to
its intersection with the penetrator.|)


(comment                      Behaviors                                  )
```

```
(ask fighter-base documentation for
     (activate) is
|Sender: filter-center. Fighter-base goes on 'alert' status for
'alert-duration' number of seconds, and then goes back to 'active'
status.|)


(ask fighter-base documentation for
     (wind down) is
|Sender: fighter-base. Sets status to 'active' (typically from 'alert'
status).|)


(ask fighter-base documentation for
     (scramble some fighters guided by >GCI to >penetrator) is
|Sender: command-center. Assigns fighters to a penetrator and makes them
unavailable for other tasks. The fighters selected must be 'available';
they are also the nearest ones to the penetrator; they also must have
adequate fuel and missiles. Note that the method for determining the
nearest fighter does not use the expected intercept point with the
penetrator, but rather the penetrator's last estimated location.|)


(ask fighter-base documentation for
     (determine which of >fighters is nearest >penetrator) is
|Sender: fighter-base. Note that this is estimated using the position of
the penetrator without updating. This is a deliberately inaccurate value
for a moving object. It is intended to be a realistic value, the only one
really known to the fighter-base.|)


(ask fighter-base documentation for
     (send >fighter guided by >GCI to >penetrator) is
|Sender: fighter-base. Tells the 'fighter' after 'scramble-delay' number
of seconds, to chase the 'penetrator' under guidance from 'GCI'. 'Fighter'
is removed from the fighter-base's list of 'fighters- available' and is
added to its list of 'fighters-scrambled'.|)


(ask fighter-base documentation for
     (>fighter unable to chase >penetrator) is
|Sender: fighter. If possible, the fighter-base scrambles more fighters
to the penetrator if possible. |)


(ask fighter-base documentation for
     (put >fighter on your list of >attrib) is
|Sender: fighter. 'Attrib' can be 'fighters-available', 'fighters-
scrambled' or 'fighters-destroyed'.|)
```

## 5.9. Fighter

```
(comment                    Overview
|Fighters are assigned by their fighter bases to chase penetrators. During
penetrator pursuit they are guided by a GCI (radar). If they locate the
penetrator, they enter an end game in which the options are win, lose or
draw. If they fail to find the penetrator (it leaves the radar coverage of
their guiding radar), then they return to base. Note that fighters then
become available for new missions, providing they have enough fuel and
arms. They need not be on the ground to be assigned.|)


(comment                          Behaviors                          )

(ask fighter documentation for
     (chase >penetrator guided by >GCI) is
|Sender: fighter's base. This message initiates fighter's action.
Fighter sets status to scrambled, then, requiring guide-time, asks the
guiding GCI to guide it to penetrator. The fighter base only assumes that
the fighter is available, not necessarily back at base.|)


(ask fighter documentation for
     (are on the ground) is
|Sender: fighter. It is on the ground if it has a zero velocity.|)


(ask fighter documentation for
     (take off) is
|Sender: fighter. At take off, fighter sets its velocity to its chase
velocity. Direction bears no relation to a penetrator until a GCI causes
it to be revectored. Fighter also records the time when it took off from
base.|)


(ask fighter documentation for
     (chase >penetrator to >position) is
|Sender: guiding radar. Tells fighter of new location at which to
intercept penetrator.|)


(ask fighter documentation for
     (follow unguided policy with >penetrator) is
|Sender: radar when unable to guide a fighter. Current policy for
unguided fighters, either because GCI is dead, blind, or saturated, is to
stop looking for penetrator and return to base.|)


(ask fighter documentation for
     (engage >penetrator) is
|Sender: guiding radar. Fighter expects to find penetrator in its range
at this time. If it does, it enters end game; if it does not, it returns
to base.|)


(ask fighter documentation for
     (>penetrator is in your range) is
```

|Sender: scheduler or guiding radar. Fighter has sighted the penetrator; notifies its guiding radar and commences end game. Currently this message is not used.|)

(ask fighter documentation for
    (commence end game with >penetrator) is
|Sender: fighter. Currently this is done by a random draw. Three possible outcomes are: penetrator killed, fighter killed or draw. If a fighter is successful, it becomes available again for a fresh mission.|)

(ask fighter documentation for
    (compute the end game result) is
|Sender: fighter. The end game result is either win, lose or draw and is computed using two factors: the fighters win/lose-probabilities and the number of missiles the fighter has left. The effective win probability is decremented 25% of its original value for the number of missiles less than 4 that the fighter has. This amount is distributed to the effective probability of lose and draw according to their original relative proportions.|)

(ask fighter documentation for
    (decrement your missiles) is
|Sender: fighter. The fighter decrements its missiles by the number it fires at the penetrator.|)

(ask fighter documentation for
    (return to base) is
|Sender: fighter or guiding radar. Note that when returning to base, fighter's status is 'available'. The base is in fact indifferent as to whether a fighter is in the air or on the ground.|)

(ask fighter documentation for
    (land) is
|Sender: fighter as it arrives at base. It resets its velocity and position to defaults for a grounded aircraft.|)

(ask fighter documentation for
    (rearm) is
|Sender: fighter. A fighter rearms on the ground by replenishing its missiles. The implementation is a bit tricky -- by 'forgetting' its missiles attribute, a fighter will recall its parent's missiles attribute (i.e., the default for the generic class fighter) when next asked.|)

(ask fighter documentation for
    (look for >object) is
|Sender: guiding radar possibly. Message no longer used. To look for a penetrator is to continue to use the proximity-detection algorithm until

the penetrator comes into the radar range of the fighter. This is basically just a pretty way to invoke the 'time-until-interaction' behavior, except it removes any previous plans it had.|)

(ask fighter documentation for
    (stop looking for >penetrator) is
|Sender: fighter or guiding radar. Unschedules time-until-interaction messages in a pretty way.|)

(ask fighter documentation for
    (>pen is out of your range) is
|Sender: fighter. The fighter stops looking for the penetrator. Refer to message (stop looking for >penetrator).|)

(ask fighter documentation for
    (has enough missiles) is
|Sender: fighter base when determining eligibility of fighter for a mission. If the number of missiles is greater than 1, fighter has enough missiles for a mission; otherwise it does not.|)

## 5.10. Command Center

(comment Overview

|Command center is currently the top-level decision maker for command and control. It receives input about penetrators from filter centers and makes decisions about which fighter bases should be assigned to attack them. Note that currently we assume that location and other information about a penetrator need not be passed to the command center.|)

(comment Behaviors )

(ask command-center documentation for
    (>penetrator monitored by >GCI is hostile) is
|Sender: filter-center. The command-center looks for a fighter-base that can scramble some fighters to chase a penetrator. The base selected must be on alert and the nearest one to the penetrator's estimated position. Note that this position may be inaccurate, and in any case, the base nearest the location might not be the base nearest the intercept point. The rationale is that the command-center at this time might not have detailed information about position or velocity of the penetrator.|)

(ask command-center documentation for
    (determine which of >bases is nearest >penetrator) is
|Sender: command-center. Returns the base nearest the "most recently known" position of the penetrator.|)

(ask command-center documentation for
    (>GCI has lost >penetrator) is
|Sender: filter-center. Command-center reacts to a lost penetrator by eliminating it from the appropriate lists.|)

## 5.11. Target

(comment Behaviors )

(tell target documentation for
    (bombs dropped at >position) is
|Sender: physicist. Sets status of target at 'position' to be 'destroyed'. This status is also used by the graphics program to display an explosion on top of the target.|)

## 5.12. Fixed Object

(comment Overview

|Fixed object is a generic class object whose generic subclasses include radar, SAM, filter-center, command-center, fighter-base, and target. They inherit the fixed-object's (lack of) speed.|)

(comment Behaviors )

(ask fixed-object documentation for
    (determine the current position of >object) is
|Sender: Any actor that may wish to access the current position of 'object'.|)

## 5.13. Moving Object

(comment Overview

|Moving-object is a superclass object whose subclasses include penetrator, fighter, and AWACS. Moving-object contains behaviors common to all these. Behaviors divide broadly into two sets. The first set comprises behaviors that allow the object to fly its pre-planned flight plan. The second set comprises behaviors for doing the "check-pair algorithm", i.e., for locating earliest times at which moving objects may intersect other objects' radar ranges. See [FAU81] for a complete description and analysis of this algorithm.|)

(comment Behaviors )

(tell moving-object documentation for
    (commence flight) is
|Sender: simulation initialization. Causes a moving object with flight plan (e.g., penetrator or AWACS) to begin flight.|)

```
(tell moving-object documentation for
      (next sector) is
|Sender:  the moving object.  Causes moving object to update its  position,
execute  its  next  sector,  and schedule the subsequent part of its flight
plan for execution.  If object is an AWACS, the  executed  flight  plan  is
appended to the existing one;  if not it is discarded.|)


(tell moving-object documentation for
      (fly to >place) is
|Sender:  the moving object.  The moving object revectors itself  from  its
current  position  to  'place'.   Note:  penetrator uses its own version of
this behavior.|)


(tell moving-object documentation for
      (revector from >position-now to >place) is
|Sender:  the moving object.  The moving object sets its x and  y  velocity
components  so  that it will intersect 'place', assuming it is currently at
'position-now'.|)


(tell moving-object documentation for
(check interaction of route from >position-now to >place with >radar)
  is
|Sender:  penetrator to itself.  Sent when penetrator is flying to a place.
If the penetrator will  enter  and  exit  the  radar's  range,  causes  the
penetrator  to  ask  scheduler to send in/out range messages to that radar.
If penetrator will only exit during this segment then penetrator must  have
revectored  inside coverage so scheduler is notified to tell the radar that
the penetrator has changed route inside radar coverage.|)


(tell moving-object documentation for
      (give your next position) is
|Sender:  the moving object.  Returns the next  location  on  the  object's
flight plan.|)


(tell moving-object documentation for
      (determine the current position of >object) is
|Sender:  the moving object.  Asks 'object'  to  update  its  position  and
return  that  new position.  Note that a direct "recall your position" need
not be accurate because position is only updated on demand.|)


(tell moving-object documentation for
      (update your position)  is
|Sender:  several objects.  Object computes its new position as a  function
of velocity and time of last update;  installs that value.|)


(tell moving-object documentation for
      (initialize interactions between >set1 >set2) is
```

|Sender: simulation initialization. Set1 and set2 are moving objects. Behavior causes objects in set1 to periodically check to see if those in set2 are within their radar range, using the check-pair algorithm. The main behaviors for this algorithm are: (time-until-interaction >o1 >o2) (test-intercept >o1 >o2) and (monitor-interaction >o1 >o2).|)

(tell moving-object documentation for
(time-until-interaction >object1 >object2) is
|Sender: the moving object. Used in check-pair computation. Object1 first computes earliest time (x) it could be in object2's radar range. If this is 0, then it issues an in range message and monitors the interaction (see (monitor-interaction >o1 >o2)). If not, it plans to check this proximity again in x seconds or MIN-CHECK-PAIR-TIME seconds, whichever is greater.|)

(tell moving-object documentation for
(test-intercept >object1 >object2) is
|Sender: the moving object. Assuming both objects are heading toward each other at maximum speed, determines earliest intersection of object1 with object2's radar range. If object2 has no radar range, this intersection is a collision.|)

(tell moving-object documentation for
(monitor-interaction >object1 >object2) is
|Sender: the moving object. Moving object iteratively schedules this behavior after object1 has been determined to be in object2's range. It continues until object1 goes out of range. If the object does go out of range, then a (time-until-interaction >object1 >object2) is scheduled. If the object does not, then a "monitor-interaction" is rescheduled. Rescheduling time is a function of how near object1 is to object2's radar boundary. MIN-MONITOR-TIME puts a lower bound on this.|)

(tell moving-object documentation for
(min-time >object1 >object2 >distance) is
|Sender: the moving object. Determines the minimum time, in seconds, in which the objects could be within 'distance' of one another.|)

(tell moving-object documentation for
(time-in-mps >distance >velocity) is
|Sender: several objects. Time, in seconds, to go 'distance' at 'velocity'.|)

(tell moving-object documentation for
(are moving) is
|Sender: the moving object. Predicate returns non-nil only if object has a non-zero velocity.|)

## 5.14. Scheduler

```
(comment                    Overview
```

|The scheduler, the mathematician and the physicist are "auxiliary objects." They are distinct from the "basic objects" in that they do not represent real-world actors present in the modelled domain. They process events that are not the direct result of any object's intentional actions. Such events include the entry into, and exit from, a radar range by a penetrator; the effect of ecm; and the turning of an aircraft inside radar coverage. The idea is to let auxiliary objects handle these non-intentional events, so that the code for real simulation objects can be written as transparently as possible. That is, we do not want to have to write behaviors for real objects that have no correlate in the real-world. Otherwise the naturalness of programming in an object-oriented style will be lost. The creation of auxiliary objects, like the scheduler, allows us to do this.

The main activity of the scheduler is to retain and distribute messages to radars concerning the entry and exit of penetrators to and from their range. For stationary radars, these entry and exit events are precomputed for every penetrator. Each is marked with a time, and the scheduler plans to issue them at that time. The issuing of the messages is contingent on several factors: (i) whether the radar is blinded; (ii) whether the radar is saturated; and (iii) whether the penetrator is invisible, due to ecm. If any of these conditions arise in range and out range messages are withheld. When the conditions subside, the scheduler sends the appropriate messages.|)

```
(comment                    Behaviors                        )

(tell scheduler documentation for
     (in >time seconds tell >GCI that >penetrator is in your range) is
```

|Sender: a penetrator. Enables scheduler to tell other objects, at the appropriate times, when something is entering a radar range. Note that the messages actually sent are contingent: if the GCI is active, then an in-range message is sent; if not, then the new penetrator is placed on a list, but not dealt with. The scheduler uses this list to set the GCI's state properly when it is no longer blinded.|)

```
(tell scheduler documentation for
 (in >time seconds tell >GCI that >penetrator is out of your range) is
```

|Sender: a penetrator. Enables scheduler to tell other objects, at the appropriate times, when something leaves a radar range. Note that the messages actually sent are contingent: if the GCI is active, then an out-range message is sent; if not, then the penetrator is taken off the list of objects visible post-blinding. The scheduler uses this list to set the GCI's state properly when it is no longer blinded.|)

```
(tell scheduler documentation for
      (tell >GCI that >penetrator has changed route) is
|Sender:  penetrator.  Sent when a penetrator plans  to  fly  a  new  route
segment  and has to calculate interactions.  The change in route is sent to
the scheduler who transmits  it,  at  the  appropriate  time,  to  impacted
radar.|)


(tell scheduler documentation for
      (>GCI is blinded) is
|Sender:  physicist.  Scheduler sets things up so that when unblinded,  the
GCI  will  immediately have the correct list of possible penetrators and be
sent the appropriate in-range  messages.   Objects-visible-after- blinding
will   be  the  new  list  of  possible-penetrators;  old-penetrators  are
remembered so  that  in-range  messages  are  sent  only  for  objects  not
previously  known (no messages in case of penetrator that was in, went out,
came back in radar coverage).  Note that nothing is done if GCI is  already
blinded when scheduler receives 'blinded' message.|)


(tell scheduler documentation for
      (>GCI is unblinded)  is
|Sender:  physicist.  Scheduler sets GCI's possible-penetrators  list,  and
sends in-range messages for visible objects that were not previously on the
radar's scope, as well as 'out-range' messages for  any  that  have  exited
coverage.|)


(tell scheduler documentation for
      (eliminate old >type range messages
       involving >object1 and >object2)    is
|Sender:  scheduler.  Allows the scheduler to unplan any activity of 'type'
that it has scheduled between 'object1' and 'object2'.|)


(tell scheduler documentation for
      (>penetrator is destroyed)  is
|Sender:   fighter  after  it  has  destroyed  a   penetrator.   Scheduler
unschedules  all  previously computed interactions involving penetrator and
anything else.|)
```

## 5.15. Mathematician

```
(comment                          Behaviors                              )

(tell mathematician documentation for
      (intersection times of >p to >pn at >speed with >radar) is
|Sender:  moving-object.  Returns a list of the form (t1 t2) where 't1'  is
the time at which a penetrator flying at 'speed' between 'p' and 'pn' comes
inside radar coverage of 'radar', and  't2'  denotes  when  the  penetrator
leaves radar coverage.  |)

(tell mathematician documentation for
     (distance from >p1 to >p2) is
|Sender:  any object.  Computes distance between 'p1' and 'p2', in miles.|)
```

```
(tell mathematician documentation for
      (is >obj heading towards >radar) is
|Sender:  filter-center.  Calculates dot product of vector between 'object'
and  'radar'  with  object's velocity vector.  Returns 'true' if product is
positive, returns 'false' otherwise.|)


(tell mathematician documentation for
      (time to go from >position-now to >place at >speed) is
|Sender:  any object.  Value of time returned is in seconds.|)


(tell mathematician documentation for
      (distance >obj in range of >radar) is
|Sender:  SAM.  Computes total distance 'object' will remain in coverage of
'radar'.   Probability  that  SAM  kills  'object'  is proportional to this
distance.|)


(tell mathematician documentation for
      (will >obj come in range of >radar) is
|Sender:  filter-center.  Determines if 'object' will come into coverage of
'radar' during current leg of flight.|)


(tell mathematician documentation for
      (determine time and position of interception of >fighter with
       >penetrator) is
|Sender:   radar  (GCI).   Calculates  intercept  point  at which 'fighter'
should fly in order to intercept 'penetrator'.|)


(tell mathematician documentation for
      (is >object in range of >radar) is
|Sender:  fighter.  If 'object' is in range of fighter's  'radar',  fighter
commences endgame.|)


(tell mathematician documentation for
      (velocity of >object after turning >n degrees in >direction) is
|Sender:  penetrator.  Calculates new  velocity  when  'object'  wishes  to
evade by turning 'n' degrees left or right.|)


(tell mathematician documentation for
      (give scaled probability that >GCI detects >penetrator) is
|Sender:  radar.  Calculates probability, scaled up by a factor of 10, that
'GCI'  will  detect  'penetrator'  that  may or may not be using electronic
counter measures.|)


(tell mathematician documentation for
      (>fighter has enough fuel to chase >penetrator to >position
       and return with safety margin >m miles)  is
|Sender:  fighter-base.  This is one criteria used by fighter-base to
select candidate fighters for scrambling.|)


(tell mathematician documentation for
     (distance >fighter has left) is
|Sender:  mathematician.  Calculates maximum distance 'fighter' can fly
given its current fuel reserves.|)


(tell mathematician documentation for
     (velocity when going from >p1 to >p2 in >time) is
|Sender:  any object.  Computes velocity required to go from 'p1' to 'p2'
in 'time'.|)


(tell mathematician documentation for
     (new position after traveling from >p at >v in >t) is
|Sender:  any object.  Computes new position from previous location 'p',
velocity 'v', and time 't'.|)
```

## 5.16. Physicist

```
(comment                    Overview

|The physicist computes the effects of phenomena such as explosions and
ecm.  When a bomb is dropped, all radars within "blind-radius" of the
explosion are rendered inoperable for "blind-time" seconds.  These effects
are calculated from the magnitude and altitude of the explosion.|)


(comment                    Behaviors                              )

(ask physicist documentation for
     (>m megaton bombs exploded at >position altitude >h) is
|Sender:  scheduler.  This simulates blinding of radars as a result of
explosions.  Penetrators disappear from radar screens.  They reappear after
the blinding time.  Blinding time and blinding radius are functions of
weight of the bomb and height at which it exploded.|)


(ask physicist documentation for
     (find >GCI within >distance of >position) is
|Sender:  physicist.  Determines if a GCI radar is within blinding radius
'distance' of center 'position'.|)


(ask physicist documentation for
     (deactivate >GCI for >duration) is
|Sender:  physicist.  Informs scheduler that 'GCI' radar is blinded and
also when it will become unblinded.|)
```

## 6.0. SWIRL CODE

In this section the actual SWIRL code defining behaviors of generic objects is presented. Each object is presented in turn, in the same order in which the documentation was presented. The section begins with a listing of the abbreviations that were used to make the code highly readable:

```
(abbreviate '(ask !myself) 'you)
(abbreviate '(ask !myself) 'me)
(abbreviate '(ask something) 'sthng)
(abbreviate '(ask !myself recall your) 'your)
(abbreviate '(ask >vl create an instance) '(an >vl))
(abbreviate '(!) 'the)
(abbreviate '(!myself) 'yourself)
(abbreviate '(&) 'execute)
(abbreviate '(setq >var >val) '(let >var be >val))
(abbreviate '(&) 'that)
(abbreviate '(ask !myself schedule after !>vl seconds) '(after >vl))
(abbreviate '(ask !myself schedule after !>vl seconds) '(requiring >vl))
(abbreviate '(ask >vl recall your offspring) '(every >vl))
(abbreviate '(trace your behavior matching) 'tybm)
(abbreviate '(prog >v) '(block with local variables >v))
```

### 6.1. Penetrator

```
(ask moving-object make penetrator with
     position          nil
     max-speed         600.0
     speed             600.0
     bombs             nil
     status            nil     ;go destroyed or mission-completed
     evade-delay       100     ;number of seconds after leaving radar
                               ;coverage that evasion begins
     evade-duration    500     ;duration of evasive turn
     target-list       nil     ;list of positions where bombs will be dropped
     dying-time        100     ;time it takes the penetrator to die after
                               ;it is shot down
     ecm-probability   0.5     ;equivalent to probability that penetrator
                               ;detects it is entering radar coverage, in
                               ;which case he turns ecm on
     flight-plan       nil)    ;its planned route


(ask penetrator when receiving (fly to >place)
     (~you set your flying-toward to ~the place)
     (~you revector from !(~your position) to ~the place)
     (foreach radar in (append (ask SAM recall your offspring)
                               (ask GCI recall your offspring))
              do
              (~you check interaction of route from !(~your position)
                    to ~the place with ~the radar))


(ask penetrator when receiving(drop >m megaton bombs exploding at altitude >h)
     (tell target bombs dropped at !(car (~your target-list)))
     (~you decrement your bombs by 1)
     (tell physicist !m megaton bombs exploded at
           !(car (~your target-list)) altitude !h)
     (~you set your target-list to !(cdr (~your target-list))))


(ask penetrator when receiving (>radar is now tracking you)
     nil)


(ask penetrator when receiving (>radar is no longer tracking you)
     nil)


(ask penetrator when receiving (evade)
     (~you plan after !(~your evade-delay) seconds make a random turn)
     (~you reschedule your next sector))


(ask penetrator when receiving (reschedule your next sector)
     (~you add !`(fly to ,(~your flying-toward)) to your list of flight-plan)
     (~you unplan all (next sector))
     (~you plan after !(~your evade-duration) seconds next sector)))


(ask penetrator when receiving (make a random turn)
     (caseq (random 2)
       (0 (~you make a turn 90 degrees right))
       (1 (~you make a turn 90 degrees left))))


(ask penetrator when receiving (make a turn >n degrees >direction)
     (~you set your velocity to
           !(ask mathematician velocity of ~yourself after turning ~the n
                 degrees in ~the direction)))
```

## 6.2. Radar

```
(ask fixed-object make radar with
     status              active  ;statuses: (actived destroyed blinded)
     reporting-delay     60      ;variable, 60 is the current guess
     range               100.0   ;variable, 100 is a good guess
     filter-center       nil     ;variable
     position            nil     ;variable
     objects-guided     nil      ;the list of fighters currently guided
     saturation-limit   10       ;maximum number of penetrators that
                                 ;the radar can handle at a time
     GCI-change-delay   30       ;how long it takes GCI to find another to
                                 ;guide a fighter
     computing-delay    30       ;how long it takes to compute various
                                 ;parameters of a moving object
     fighter-communication-time
                        120      ;how frequently GCI guides fighter
     detection-probability-with-ecm
                        0.1      ;probability that radar can detect a pen
                                 ;whose ecm is on
     detection-probability-without-ecm
                        0.9      ;probability that radar can detect a pen
                                 ;whose ecm is not on
                                 )


(ask radar when receiving (>thing is in your range)
     (if (and (~you are not ECMed out by ~the thing)
              (equal (~your status) 'active))
         then
         (if (~you are saturated)
             then (~you set your objects-visible-post-saturation to
                        !(append (~your objects-visible-post-saturation)
                                 (list thing)))
             else (~you set your detecting to t)
                  (~you add ~the thing to your list of possible-penetrators)
                  (~you transmit to your filter-center that
                        ~the thing is in range of !myself)
                  (if (~you are SAM) then
                      (~you fire !(ask ~yourself shots at ~the thing)
                            times at ~the thing)))))


(ask radar when receiving (transmit to your filter-center that +message)
     (~you plan after !(~your reporting-delay) seconds
                tell !(~your filter-center) ~that message))


(ask radar when receiving (>penetrator is out of your range)
     (if (equal (~your status) 'active)
         then
         (if (memq penetrator (~your objects-visible-post-saturation))
             then
             (~you remove ~the penetrator from your list of
                   objects-visible-post-saturation)
             else
             (if (memq penetrator (~your possible-penetrators))
                 then
                 (~you remove ~the penetrator from your list
                       of possible-penetrators)
                 (if (not (~your possible-penetrators)) then
                     (~you set your detecting to nil))
                 (loop for fighter in (~your objects-guided)
                       do (~you try to change guider of ~the fighter
                                to ~the penetrator))
                 (~you transmit to your filter-center that
                       !myself has lost ~the penetrator)
                 (~you check for new penetrators)))))


(ask radar when receiving
     (try to change guider of >fighter to >penetrator)
     (loop with newgci = nil
           for fighter in (~your objects-guided) do
           (~you stop guiding ~the fighter)
           (~you unplan all (+ ~the fighter chase ~the penetrator +))
           (~let newgci be (~you find a new GCI to guide ~the fighter
                                 to ~the penetrator))
           (if newgci
               then (~after (~your GCI-change-delay) tell ~the newgci
                            guide ~the fighter to ~the penetrator)
                    (~you unplan all (+ ~the fighter engage ~the penetrator))
               else (tell ~the fighter follow unguided policy with
                          ~the penetrator))))


(ask radar when receiving (find a new GCI to guide >fighter to >penetrator)
     (loop for GCI in (~every GCI)
           when (and (not (eq myself GCI))
                     (eq (ask ~the GCI recall your status) 'active)
                     (memq penetrator (ask ~the GCI recall your
                                                possible-penetrators)))
           return GCI))


(ask radar when receiving (is >penetrator still in your range)
     (and (eq 'active (~your status))
          (memq penetrator (~your possible-penetrators))))


(ask radar when receiving (guide >fighter to >penetrator)
     (~block with local variables (newgci position interception)
       (if (eq 'destroyed (~your status))
           then (tell ~the fighter follow unguided policy with ~the penetrator)
                (return nil))
       (if (not (memq penetrator (~your possible-penetrators)))
           then (~you try to change guider of ~the fighter to ~the penetrator)
                (return nil))
       (if (eq 'blinded (~your status))
           then (~you try to change guider of ~the fighter to ~the penetrator)
                (return nil))
       (if (not (memq fighter (~your objects-guided)))
           then (~you add ~the fighter to your list of objects-guided)
                (tell ~the fighter set your GCI to !myself))
       (~let interception be (ask mathematician determine time and position of
                                  interception of ~the fighter with
```

```
                                     ~the penetrator))
       (~let position be (cdr interception))
       (~you unplan all (+ ~the fighter chase ~the penetrator +))
       (~you plan after !(~your computing-delay) seconds
             ask ~the fighter chase ~the penetrator to ~the position)
       (~you unplan all (+ ~the fighter engage ~the penetrator))
       (~you plan after !(car interception) seconds tell ~the fighter
             engage ~the penetrator)))


(ask radar when receiving (stop guiding >fighter)
     (~you unplan all (guide ~the fighter +)))


(ask radar when receiving (>fighter has sighted >penetrator)
    (~you stop guiding ~the fighter))


(ask radar when receiving (>fighter unable to chase >penetrator)
     (~you unplan all (+ ~the fighter engage ~the penetrator)))


(ask  radar when receiving (are saturated)
      (if (equal (length (~your possible-penetrators))
                 (~your saturation-limit))
          then t else nil))


(ask radar when receiving (are not ECMed out by >penetrator)
     (if (lessp (random 10)
                (ask mathematician give scaled probability
                     that ~yourself detects ~the penetrator))
         then t else nil))


(ask radar when receiving (check for new penetrators)
     (if (~your objects-visible-post-saturation)
         then
         (~block with local variables (new-pen)
                 (~let new-pen be
                       (car (~your objects-visible-post-saturation)))
                 (~you remove ~the new-pen
                       from your list of objects-visible-post-saturation)
                 (tell ~yourself ~the new-pen is in your range))))


(ask radar when receiving (>penetrator is destroyed)
     (if (equal (~your status) 'active)
         then
         (~you unplan all (tell + ~the penetrator is in your range))
         (loop for object in (~your objects-guided)
               when (eq penetrator
                        (ask !object recall your penetrator-pursued))
               do (tell ~the object return to base))
```

```
        (if (or (memq penetrator (~your possible-penetrators))
                (memq penetrator (~your objects-visible-post-saturation)))
            then (tell !myself ~the penetrator is out of your range))))


(ask radar when receiving (>penetrator has changed route)
     (if (eq (~your status) 'active)
         then (loop for fighter in (~your objects-guided)
                    when (equal (ask ~the fighter recall your
                                     penetrator-pursued)
                                penetrator)
                    do (~you guide ~the fighter to ~the penetrator))))
```

## 6.3. GCI

```
(ask radar make GCI with
     status              active
     filter-center       nil
     position            nil
     objects-guided      nil)
```

## 6.4. AWACS

```
(ask radar make AWACS with
     filter-center              nil     ;who the AWACS reports detections to
     objects-guided             nil     ;who AWACS is currently guiding
     possible-penetrators       nil     ;things it is now detecting
     position                   nil)


(ask moving-object make AWACS with
     velocity                   nil     ;how fast AWACS is currently going
     max-speed                  400     ;max for checkpair
     speed                      300
     flight-plan                nil     ;its (repeated) route
     time                       0)      ;scratch attribute for determining loc


(ask AWACS when receiving (look for >penetrator)
     (~you unplan all (time-until-interaction ~the penetrator !myself))
     (~you time-until-interaction ~the penetrator !myself))
```

## 6.5. SAM

```
(ask radar make SAM with
     position              (nil nil)
     status                active
     range                 50
     alert-delay           200
     computing-delay       0
     missiles              nil
     max-shots              4
     possible-penetrators   nil
     filter-center          nil)


(tell SAM when receiving (expect >penetrator)
    (~you set your status to alert)
    (~you add ~the penetrator to your list of possible-penetrators))


(tell SAM when receiving (>pen has changed route)
      nil)

(tell SAM when receiving (fire at >penetrator)
      (if (and (memq penetrator (~your possible-penetrators))
               (car (~your missiles))) then
          (~block with local variables (interception position)
            (~let missile be (car (~your missiles)))
            (~you add ~the missile to your list of missiles-launched)
            (~let interception be (ask mathematician
                                       determine time and position of
                                       interception of
                                       ~the missile with ~the penetrator))
            (~let position be (cdr interception))
            (~you plan after !(~your computing-delay) seconds
                  ask ~the missile chase ~the penetrator to ~the position)
            (~you plan after !(car interception) seconds tell ~the missile
                  hit ~the penetrator)
            (~you set your missiles to !(cdr (~your missiles))))
          else
          (tell !(~your filter-center) ~yourself missed ~the penetrator)))


(tell SAM when receiving (>missile destroyed >penetrator)
      (tell scheduler ~the penetrator is destroyed))

(tell SAM when receiving (>missile missed >penetrator)
      (if (ask ~yourself is ~the penetrator still in your range)
          then
          (~you fire at ~the penetrator)))
```

## 6.6. Missile

```
(ask moving-object make missile with
     position          nil          ; position, should begin as SAM's
     velocity          (0.0 0.0)    ; missile current velocity
     chase-velocity    (710. 0.0)   ; missile chase velocity,
                                    ; as velocity-pair for uniformity
     speed             710.0        ; handy to have around even if redundant
     range             0.0
     status            nil
     exploding-time    !(ask nclock recall your $ticksize)
     SAM               nil)         ; SAM which fires it
```

```
(ask missile when receiving (chase >penetrator to >position)
     (~you set your velocity to !(~your chase-velocity))
     (~you update your position)
     (~you revector from !(~your position) to ~the position))


(ask missile when receiving (hit >penetrator)
     (if (memq (random 4) '(0 1 2 3)) then  ;kill probability 0.0

         (tell !(~your SAM) ~yourself missed ~the penetrator)
         (~you set your velocity to (0.0 0.0))
         (~you set your status to exploding)
         (~you plan after !(~your exploding-time) seconds
               set your status to destroyed)
         else
         (tell !(~your SAM) ~yourself destroyed ~the penetrator)
         (~you set your velocity to (0.0 0.0))
         (~you set your status to exploding)
               'setting 'my 'status 'to (ask !myself recall your status)))
         (~you plan after !(~your exploding-time) seconds
               set your status to destroyed)))
```

## 6.7. Filter Center

```
(ask fixed-object make filter-center with
     position           nil
     status             active  ;active or destroyed
     reporting-delay    70      ;the time it takes to interact with cc
     computing-delay    90      ;the time it takes to determine hostility
     alerting-delay     60      ;the time to alter bases
     command-center     nil     ;who fc reports to
     fighter-bases      nil     ;available bases
     SAMs               nil     ;available SAMs
     GCIs               nil     ;reporting GCIs
     penetrators        nil)    ;penetrators tracked


(ask filter-center when receiving (>thing is in range of >GCI)
     (if (~you determine ~the thing is a hostile penetrator)
         then (~after (~your computing-delay) ~you prepare to defend
                          against ~the thing monitored by ~the GCI)))


(ask filter-center when receiving (determine >thing is a hostile penetrator)
     t)


(ask filter-center when receiving
     (prepare to defend against >thing monitored by >GCI)
     (if (ask ~the GCI is ~the thing still in your range)
        then (~you add ~the thing to your list of penetrators)
             (~you alert your fighter-bases about ~the thing)
             (~you alert your SAMs about ~the thing)
             (~you transmit to your command-center that
                  ~the thing monitored by ~the GCI is hostile)))


(ask filter-center when receiving (alert your fighter-bases about >penetrator)
     (loop for fighter-base in (~your fighter-bases)
           when (and (ask mathematician is ~the penetrator heading
                          towards ~the fighter-base)
                     (ask mathematician will ~the penetrator
                          come in range of ~the fighter-base))
           do (~you transmit to ~the fighter-base to activate)))


(ask filter-center when receiving
     (alert your SAMs about >penetrator)
     (loop for SAM in (~your SAMs)
           when (and (ask mathematician is ~the penetrator heading
                          towards ~the SAM)
                     (ask mathematician will ~the penetrator
                          come in range of ~the SAM))
           do (~you transmit to ~the SAM
                    to expect ~the penetrator)))


(ask filter-center when receiving (transmit to >agent to +directive)
     (~requiring (~your alerting-delay) tell ~the agent ~that directive))


(ask filter-center when receiving
     (transmit to your command-center that +message)
     (~requiring (~your reporting-delay) tell !(~your command-center)
                 ~that message))


(ask filter-center when receiving (>GCI has lost >penetrator)
     (~you remove ~the penetrator from your list of penetrators)
     (tell !(~your command-center) ~the GCI has lost ~the penetrator))


(ask filter-center when receiving (>penetrator killed by >agent)
     (~you remove ~the penetrator from your list of penetrators))


(ask filter-center when receiving (>agent missed >penetrator)
     nil)
```

## 6.8. Fighter Base

```
(ask fixed-object make fighter-base with
      position                  (nil nil)       ;position
```

```
     status                    active          ;active or destroyed
     filter-center             nil             ;each base has one
     fighters-available        nil             ;list of base's free fighters
     fighters-scrambled        nil             ;occupied fighters
     fighters-destroyed        nil             ;its fighters lost
     scramble-delay            10              ;time to scramble once told
     alert-delay               10              ;delay to alert
     alert-duration            1800            ;how long to remain alert
     range                     400.0)          ;how far its fighters can go


(ask fighter-base when receiving (activate)
    (~requiring (~your alert-delay) set your status to alert)
    (~you unplan all (wind down))
    (~after (~your alert-duration) wind down))


(ask fighter-base when receiving (wind down)
     (~you set your status to active))


(ask fighter-base when receiving
   (scramble some fighters guided by >GCI to >penetrator)
     (if (eq (~your status) 'alert)
         then
         (loop with fighters = (~your fighters-available) and candidates = nil
               while fighters
               do (~let candidate be (~you determine which of ~the fighters is
                                            nearest ~the penetrator))
                  (if (and (ask ~the candidate has enough missiles)
                           (ask mathematician ~the candidate has enough fuel
                                to chase ~the penetrator to
                                !(ask ~the penetrator recall your position)
                                and return with safety margin 200 miles))
                      then (~you send ~the candidate guided by ~the GCI to
                                 ~the penetrator)
                           (return nil)
                      else (~let fighters be (delete candidate fighters))))))


(ask fighter-base when receiving
  (determine which of >fighters is nearest >penetrator)
     (loop with distance = nil and minimum-distance = 10000 and
                nearest-object = nil
           for object in fighters
           do (~let distance be (ask mathematician distance from
                                   !(ask ~the object recall your position) to
                                   !(ask ~the penetrator recall your position)))
           when (lessp distance minimum-distance)
           do (~let minimum-distance be distance)
              (~let nearest-object be object)
           finally (return nearest-object)))
```

```
(ask fighter-base when receiving (send >fighter guided by >GCI to >penetrator)
     (~you schedule after !(~your scramble-delay) seconds
           tell ~the fighter chase ~the penetrator guided by ~the GCI)
     (~you add ~the fighter to your list of fighters-scrambled)
     (~you remove ~the fighter from your list of fighters-available))


(ask fighter-base when receiving (>fighter unable to chase >penetrator)
     (~you scramble some fighters guided by !(ask ~the fighter recall your GCI)
           to ~the penetrator))


(ask fighter-base when receiving (put >fighter on your list of >attrib)
     (~you set your !attrib to !(append (~your !attrib) (list fighter))))
```

## 6.9. Fighter

```
(ask moving-object make fighter with
     position            nil                 ; position, should begin as base's
     velocity            (0.0 0.0)           ; fighter current velocity
     chase-velocity      (710.0 0.0)         ; fighter chase velocity, given as
                                             ; x y vector
     speed               710.0               ; handy to have around but redundant
     max-speed           710.0
     range               30.0                ; range of onboard radar
     status              nil                 ; scrambled, available or destroyed
     base                nil                 ; its home base
     missiles            6                   ; default number of missiles
     penetrator-pursued  nil                 ; who the fighter is aiming at
     fuel                6000                ; amount of fuel
     mpg                 0.55                ; miles per gallon
                                             ; (California rating may be lower)
     win-probability     0.0                 ; probability a fighter beats a pen
     lose-probability    1.0                 ; probability a pen beats a fighter
     time-of-departure   nil                 ; time fighter departs from base
     dying-time          100                 ; time it takes the fighter to die
                                             ; after it is shot down
     guide-time          40                  ; time to get guidance request to GCI
     GCI                 nil)                ; current GCI that is guiding fighter
                                             ; when scrambled


(ask fighter when receiving (chase >penetrator guided by >GCI)
     (~you unplan all (land))
     (~you set your status to scrambled)
     (if (~you are on the ground) then (~you take off))
     (~requiring (~your guide-time) tell ~the GCI guide !myself
                to ~the penetrator))


(ask fighter when receiving (are on the ground)
     (equal '(0.0 0.0) (~your velocity)))
```

```
(ask fighter when receiving (take off)
     (~you set your velocity to !(~your chase-velocity))
     (~you set your time to !(ask nclock recall your $stime))
     (~you set your time-of-departure to !(ask nclock recall your $stime)))


(ask fighter when receiving (chase >penetrator to >position)
         (~you set your penetrator-pursued to ~the penetrator)
         (~you update your position)
         (~you revector from !(~your position) to ~the position))


(ask fighter when receiving (follow unguided policy with >penetrator)
     nil)


(ask fighter when receiving (engage >penetrator)
     (if (and (ask mathematician is ~the penetrator in range of ~yourself)
              (not (eq (~your status) 'destroyed)))
         then (~you commence end game with ~the penetrator)
         else (~you return to base)))


(ask fighter when receiving (>penetrator is in your range)
     (~you stop looking for ~the penetrator)
     (tell !(~your GCI) ~yourself has sighted ~the penetrator)
     (~you commence end game with ~the penetrator))


(ask fighter when receiving (commence end game with >penetrator)
     (~you set your detecting to t)
     (caseq (~ you compute the end game result)
       (lose
        (~you set your status to dying)
        (~you plan after !(~your dying-time) seconds set your
              status to destroyed)
        (~you set your velocity to (0.0 0.0))
        (tell !(~your base) remove !myself from your list
              of fighters-scrambled)
        (tell !(~your base) add !myself to your list
              of fighters-destroyed))
       (draw
        (~you decrement your missiles)
        (~you return to base))
       (win
        (~you decrement your missiles)
        (tell scheduler ~the penetrator is destroyed)
        (~you return to base)))
     (~you set your detecting to nil))


(ask fighter when receiving (compute the end game result)
     (let* ((win-probability
          (difference (~your win-probability)
                      (times (~your win-probability)
                             (times .25
                                    (- 4 (~your missiles))))))
         (lose-probability
          (plus (~your lose-probability)
                (times (~your win-probability)
                   (times (times .25
                                 (quotient (~your lose-probability)
                                           (difference 1.0
                                             (~your win-probability))))
                          (- 4 (~your missiles))))))
         (win-number (fix (times 100 win-probability)))
         (lose-number (+ win-number (fix (times 100 lose-probability))))
         (number (random 100)))
    (cond ((< number win-number) 'win)
          ((< number lose-number) 'lose)
          (t 'draw))))


(ask fighter when receiving (decrement your missiles)
     (let ((number (1+ (random 6))))
       (if (> number (~your missiles))
           then (~you set your missiles to 0)
           else (~you decrement your missiles by ~the number))))


(ask fighter when receiving (return to base)
     (~you update your position)
     (~you revector from !(~your position)
           to !(ask !(~your base) recall your position))
     (~you set your status to available)
     (~you set your penetrator-pursued to nil)
     (ask !(~your base) put !myself on your list of fighters-available)
     (~you schedule after !(ask mathematician time to go from !(~your position)
                                 to !(ask !(~your base) recall your position)
                                 at !(~your speed))
           seconds land))


(ask fighter when receiving (land)
     (~you set your velocity to (0.0 0.0))
     (~you rearm)
     (ask !(~your base) remove !myself from your list of fighters-scrambled)
     (~you set your position to !(ask !(~your base) recall your position)))


(ask fighter when receiving (rearm)
     (~you forget your missiles))


(ask fighter when receiving (look for >object)
     (~you unplan all (time-until-interaction ~the penetrator !myself))
     (~you time-until-interaction ~the penetrator !myself))
```

```
(ask fighter when receiving (stop looking for >penetrator)
     (~you unplan all (time-until-interaction ~the penetrator !myself)))


(ask fighter when receiving (>pen is out of your range)
     (~you stop looking for ~the pen))


(ask fighter when receiving (has enough missiles)
     (> (~your missiles) 1))
```

## 6.10. Command Center

```
(ask fixed-object make command-center with
     fighter-bases        nil      ;accessible fighter bases
     filter-centers       nil      ;reporting filter-centers
     position             nil      ;location
     status               active   ;active or destroyed
     command-delay        10       ;time it takes to tell fb to scramble
     penetrators-handled           ;when a fighter base is found to scramble
                          nil      ;fighters to the pen
     penetrators-not-handled       ;when no such base can be found
                          nil)


(ask command-center when receiving (>penetrator monitored by >GCI is hostile)
     (loop with fbases = nil and fbase = nil
           for fighter-base in (~your fighter-bases)
           when (and (eq (ask ~the fighter-base recall your status) 'alert)
                     (ask ~the fighter-base recall your fighters-available)
                     (ask mathematician will ~the penetrator
                          come in range of ~the fighter-base))
           collect fighter-base into fbases
           finally (if (not fbases)
                       then (~you add ~the penetrator to your list
                                  of penetrators-not-handled)
                       else (~let fbase be (~you determine which of ~the fbases
                                                 is nearest ~the penetrator))
                            (~you add ~the penetrator to your list
                                  of penetrators-handled)
                            (~requiring (~your command-delay) tell ~the fbase
                                 scramble some fighters guided by ~the GCI
                                 to ~the penetrator))))


(ask command-center when receiving
  (determine which of >bases is nearest >penetrator)
    (loop with distance = nil and minimum-distance = 10000 and
               nearest-object = nil
          for object in bases
          do (~let distance be (ask mathematician distance from
                                       !(ask ~the object recall your position) to
                                       !(ask ~the penetrator recall your position)))
             when (lessp distance minimum-distance)
             do (~let minimum-distance be distance)
                (~let nearest-object be object)
             finally (return nearest-object)))


(ask command-center when receiving (>GCI has lost >pen)
     (if (not (~you remove ~the penetrator from your list
                    of penetrators-handled))
         then (~you remove ~the penetrator from your list
                    of penetrators-not-handled)))
```

## 6.11. Target

```
(ask fixed-object make target with
     targets-to-be-destroyed nil
     targets-destroyed nil)


(ask target when receiving (bombs dropped at >position)
     (foreach targ in (~your targets-to-be-destroyed)
              do
              (if (equal (ask !targ recall your position)
                         position)
                  then
                  (ask !targ set your status to destroyed)
                  (~you add !targ to your list of targets-destroyed)
                  (~you remove !targ from your list
                        of targets-to-be-destroyed))))
```

## 6.12. Fixed Object

```
(ask simulator make fixed-object with
     max-speed          0.0)


(ask fixed-object when receiving (determine the current position of >object)
     (tell ~the object update your position)
     (ask ~the object recall your position))
```

## 6.13. Moving Object

```
(ask simulator make moving-object with
     velocity                (600.0 600.0) ;initial velocity
     sectortime              0.0           ;hours to fly current sector
     time                    0.0           ;time of last position update
     min-check-pair-time     4             ;max frequency of checkpair
     min-monitor-time        2)            ;max frequency of checkpair
                                           ;during in range
```

```
(ask moving-object when receiving (commence flight)
     (~you set your time to !(ask nclock recall your $stime))
     (tell ~yourself next sector))


(ask moving-object when receiving (next sector)
     (~block with local variables (sector)
        (if (equal (~your status) 'destroyed) then (return nil))
        (~you update your position)
        (~you set your sectortime to 0)
        (~let sector be (car (~your flight-plan)))
        (~you set your flight-plan to !(cdr (~your flight-plan)))
        (~you ~execute sector)
        (if (~you are AWACS) then
            (~you set your flight-plan to
                  !(append (~your flight-plan) (list sector))))
        (if (~your flight-plan) then
            (~you schedule after !(times (~your sectortime) 3600) seconds
                  next sector))))


(ask moving-object when receiving (fly to >place)
    (~you revector from !(~your position) to ~the place))


(ask moving-object when receiving (revector from >position-now to >place)
     (if (not (equal position-now place)) then
         (~you set your sectortime to
               !(quotient (ask mathematician distance from ~the position-now
                                to ~the place)
                          (~your speed)))
         (~you set your velocity to !(ask mathematician velocity when
                                          going from ~the position-now to
                                          ~the place in !(~your sectortime)))))


(ask moving-object when receiving
     (check interaction of route from >position-now to >place with >radar)
     (~block with local variables (intersections)
        (~let intersections be
              (ask mathematician intersection times of ~the position-now
                   to ~the place at !(~your speed) with ~the radar))
        (if (car intersections) then                        ;enters range
            (ask scheduler in !(car intersections) seconds
                 tell ~the radar that !myself is in your range)
        (if (cadr intersections) then                       ;leaves radar range
            (ask scheduler in !(cadr intersections) seconds
                 tell ~the radar that !myself is out of your range)
        (if (and (not (ask ~the radar are you a SAM))
                 (not (car intersections))
                 (cadr intersections))
            then                          ;changes route inside radar coverage
            (ask scheduler tell ~the radar that !myself has changed route))))


(ask moving-object when receiving (give your next position)
     (loop with plan = (ask !myself recall your flight-plan)
           when (null plan)
              return (ask !myself recall your position))
           else when (eq (caar plan) 'fly)
                return (caddar plan)
                else do (~let plan be (cdr plan)))


(ask moving-object when receiving (determine the current position of >object)
     (tell ~the object update your position)
     (ask ~the object recall your position))


(tell moving-object when receiving (update your position)
      (~block with local variables (simtime)
         (if (not (~you are moving)) then (return nil))
         (~let simtime be (ask nclock recall your $stime))
         (tell !myself set your position to
               !(ask mathematician new position after traveling from
                     !(~your position) at !(~your velocity) in
                     !(difference simtime (~your time))))
         (tell !myself set your time to ~the simtime)))


(tell moving-object when receiving
      (initialize interactions between >set1 >set2)
      (loop for obj1 in set1
            do (loop for obj2 in set2
                     do (ask !myself time-until-interaction !obj1 !obj2))))


(tell moving-object when receiving
      (initialize interaction between >set1 >object)
      (loop for obj in set1
            do (ask !myself time-until-interaction !obj !object)))


(tell moving-object when receiving (time-until-interaction >obj1 >obj2)
      (~block with local variables (m)
         (~let m be (ask !myself test-intercept !obj1 !obj2))
         (if m then
             (if (= m 0.0)
                 then (tell !obj2 !obj1 is in your range)
                      (ask !myself monitor-interaction !obj1 !obj2)
                 else (~let m be
                            (max m (ask moving-object recall your
                                        min-check-pair-time)))
                      (ask !myself schedule after !m seconds
                           time-until-interaction !obj1 !obj2)))))
```

```
(tell moving-object when receiving (test-intercept >obj1 >obj2)
      (ask !obj1 update your position)
      (ask !obj2 update your position)
      (if (ask !obj2 recall your range)
          then (ask moving-object min-time !obj1 !obj2
                    !(ask !obj2 recall your range))
          else (ask moving-object min-time !obj1 !obj2 0.0)))


(tell moving-object when receiving (monitor-interaction >pen >radar)
      (~block with local variables (dist-between n)
         (tell ~the pen update your position)
         (tell ~the radar update your position)
         (~let dist-between be (ask mathematician distance from
                                  !(ask ~the pen recall your position) to
                                  !(ask ~the radar recall your position)))
         (if (> dist-between (ask ~the radar recall your range))
             then (tell ~the radar ~the pen is out of your range)
                  (ask !myself schedule after 5 seconds time-until-interaction
                       ~the pen ~the radar)
                  else (~let n be
                              (ask moving-object time-in-mps
                                   !(difference (ask ~the radar recall your
                                                     range)
                                                dist-between)
                                   !(plus (ask ~the pen recall your max-speed)
                                        (ask ~the radar recall your max-speed))))
                       (~let n be (1+ (fix n)))
                       (ask !myself schedule after !(max n (ask moving-object
                            recall your min-monitor-time)) seconds
                            monitor-interaction ~the pen ~the radar))))


(tell moving-object when receiving (min-time >obj1 >obj2 >distance)
      (~block with local variables (d v)
         (~let d be (ask mathematician distance from
                         !(ask !obj1 recall your position)
                         to !(ask !obj2 recall your position)))
         (~let v be (float (plus (ask !obj1 recall your max-speed)
                                 (ask !obj2 recall your max-speed))))
         (return (if (< d distance) then 0.0
                     else (if (= v 0.0) then nil
                              else (ask moving-object time-in-mps
                                        !(difference d distance) !v))))))


(tell moving-object when receiving (min-time2 >obj >distance)
      (~block with local variables (v)
         (~let v be (float (ask ~the obj recall your max-speed)))
         (return (if (= v 0.0) then 0.0
                     else (ask moving-object time-in-mps ~the distance
                               ~the v)))))
```

```
(tell moving-object when receiving (time-in-mps >distance >velocity)
      (//$ distance (quotient velocity 3600.0)))


(tell moving-object when receiving (are moving)
      (not (equal (~your velocity) '(0.0 0.0))))
```

## 6.14. Scheduler

```
(ask simulator make scheduler)


(ask scheduler when receiving
    (in >time seconds tell >GCI that >penetrator is in your range)
    (if (not (ask ~the GCI is ~the penetrator still in your range))
        then
        (~you eliminate old in range messages involving ~the GCI
              and ~the penetrator)
        (~you schedule after ~the time seconds
              (if (eq (ask ~the GCI recall your status) 'active)
                  then (tell ~the GCI ~the penetrator is in your range)
                  else (tell ~the GCI add ~the penetrator to your list of
                             objects-visible-post-blinding))))
    (~you plan after ~the time seconds tell ~the penetrator ~the GCI
          is now tracking you))


(ask scheduler when receiving
    (in >time seconds tell >GCI that >penetrator is out of your range)
     (~you eliminate old out range messages involving ~the GCI
           and ~the penetrator)
     (~you schedule after ~the time seconds
           (~block with local variables nil
            (if (equal (ask ~the GCI recall your status) 'active)
                then (tell ~the GCI ~the penetrator is out of your range)
                else (tell ~the GCI remove ~the penetrator from your list of
                           objects-visible-post-blinding))))
     (~you plan after ~the time seconds tell ~the penetrator ~the GCI
           is no longer tracking you))


(ask scheduler when receiving (tell >GCI that >penetrator has changed route)
     (if (equal (ask ~the GCI recall your status) 'active)
         then (tell ~the GCI ~the penetrator has changed route)))


(ask scheduler when receiving (>GCI is blinded)
     (ask ~the GCI set your detecting to nil)
     (ask ~the GCI set your objects-visible-post-blinding to
          !(~your possible-penetrators))
     (ask ~the GCI set your old-penetrators to !(~your possible-penetrators))
     (ask ~the GCI set your possible-penetrators to nil))
```

```
(ask scheduler when receiving (>GCI is unblinded)
     (ask ~the GCI set your possible-penetrators to
          (~your objects-visible-post-blinding))
     (if (ask ~the GCI recall your objects-visible-post-blinding)
         then (ask ~the GCI set your detecting to t))
     (loop for object in (ask ~the GCI recall your
                                    objects-visible-post-blinding)
           when (not (memq object (~your old-penetrators)))
           do (tell ~the GCI ~the object is in your range)))


(ask scheduler when receiving
     (eliminate old >type range messages involving >GCI and >penetrator)
     (loop for item in (~your things-to-do)
           when (match `(tell ,GCI ,penetrator + ,type +)
                       (range-clause item))
           do (~you remove ~the item from your list of things-to-do))))


(ask scheduler when receiving (>penetrator is destroyed)
     (~you eliminate old in range messages involving  + and ~the penetrator)
     (~you eliminate old out range messages involving + and ~the penetrator)
     (ask ~the penetrator set your status to dying)
     (ask ~the penetrator plan after !(ask ~the penetrator recall your
           dying-time) seconds set your status to destroyed)
     (ask ~the penetrator set your velocity to (0.0 0.0))
     (loop for radar in (append (~every AWACS) (~every GCI) (~every SAM))
           do (tell ~the radar ~the penetrator is destroyed)))
```

## 6.15. Mathematician

```
(ask simulator make mathematician)


(ask mathematician when receiving (intersection times of >p to >pn
                                                at >speed with >radar)
     (~block with local variables (x y xn yn gx gy rr)
       (~let x be (car p))
       (~let y be (cadr p))
       (~let xn be (car pn))
       (~let yn be (cadr pn))
       (~let gx be (car (ask !radar recall your position)))
       (~let gy be (cadr (ask !radar recall your position)))
       (~let rr be (ask !radar recall your range))
       (return (intersection gx gy rr x y xn yn (quotient speed 3600.0)))))


(ask mathematician when receiving (distance from >p1 to >p2)
     (distance (car p1) (cadr p1) (car p2) (cadr p2)))
```

```
(ask mathematician when receiving (is >obj heading towards >radar)
     (~block with local variables (x y vx vy rx ry)
       (~let rx be (car (ask !radar recall your position)))
       (~let ry be (cadr(ask !radar recall your position)))
       (~let x be (car (ask !obj recall your position)))
       (~let y be (cadr(ask !obj recall your position)))
       (~let vx be (car (ask !obj recall your velocity)))
       (~let vy be (cadr (ask !obj recall your velocity)))
       (if (greaterp (plus (times vx (difference rx x))
                           (times vy (difference ry y))) 0.0)
           then (return t) else (return nil))))


(ask mathematician when receiving
   (time to go from >position-now to >place at >speed)
     (times 3600. (quotient (ask mathematician distance from ~the position-now
                                 to ~the place)
                             speed)))


(ask mathematician when receiving (distance >obj in range of >radar)
     (~block with local variables nil (return (distance-in-range
                       (car (ask !radar recall your position))
                       (cadr(ask !radar recall your position))
                       (ask !radar recall your RANGE)
                       (car (ask !obj recall your position))
                       (cadr(ask !obj recall your position))
                       (car (ask !obj recall your flying-toward))
                       (cadr (ask !obj recall your flying-toward))
                       (quotient (ask !obj recall your speed) 3600.0)))))


(ask mathematician when receiving (will >obj come in range of >radar)
     (if (ask mathematician distance !obj in range of !radar)
         then t else nil))


(ask mathematician when receiving (determine time and position of interception
                                    of >fighter with >penetrator)
     (~block with local variables (px py vx vy fx fy fvel)
       (ask ~the fighter update your position)
       (ask ~the penetrator update your position)
       (~let px be (car (ask ~the penetrator recall your position)))
       (~let py be (cadr (ask ~the penetrator recall your position)))
       (~let vx be (car (ask ~the penetrator recall your velocity)))
       (~let vy be (cadr (ask ~the penetrator recall your velocity)))
       (~let fx be (car (ask ~the fighter recall your position)))
       (~let fy be (cadr (ask ~the fighter recall your position)))
       (~let fvel be (ask ~the fighter recall your speed))
       (return (intercept px py (quotient vx 3600.0) (quotient vy 3600.0)
                          fx fy (quotient fvel 3600.0)))))


(ask mathematician when receiving (is >object in range of >radar)
     (ask ~the object update your position)
     (not (outside (car (ask ~the radar recall your position))
                   (cadr (ask ~the radar recall your position))
                   (ask ~the radar recall your range)
                   (car (ask ~the object recall your position))
                   (cadr (ask ~the object recall your position)))))


(ask mathematician when receiving
     (velocity of >object after turning >n degrees in >direction)
      (ask ~the object update your position)
      (turn (car (ask ~the object recall your velocity))
            (cadr (ask ~the object recall your velocity))
            (cond ((eq direction 'left) n)
                  ((eq direction 'right) (- n))
                  (t (error '"turn -- bad direction")))))


(ask mathematician when receiving
     (give scaled probability that >GCI detects >penetrator)
     (~block with local variables (prob)
       (~let prob be
             (fix
              (times 10.0
                     (plus (times (ask ~the penetrator recall your
                                       ecm-probability)
                                  (ask ~the GCI recall your
                                       detection-probability-with-ecm))
                           (times (difference 1 (ask ~the penetrator recall
                                                    your ecm-probability))
                                  (ask ~the GCI recall your
                                       detection-probability-without-ecm))))))
       (return prob)))


(ask mathematician when receiving
     (>fighter has enough fuel to chase >penetrator to >position and return
               with safety margin >m miles)
     (ask ~the fighter update your position)
     (~block with local variables (dist-to-go dist-left)
       (~let dist-to-go be (plus (ask mathematician distance from
                                    !(ask ~the fighter recall your position)
                                    to ~the position)
                                (ask mathematician distance from ~the position
                                    to !(ask ~the fighter ask your base
                                             to recall your position))))
       (~let dist-left be (ask mathematician distance ~the fighter has left))
       (if (greaterp dist-left (plus m dist-to-go)) then (return t)
           else (return nil))))


(ask mathematician when receiving (distance >fighter has left)
     (ask ~the fighter update your position)
     (if (equal (ask ~the fighter recall your status) 'scrambled)
         then
         (difference (times (ask ~the fighter recall your fuel)
                             (ask ~the fighter recall your mpg))
                     (times (difference (ask nclock recall your $stime)
                                        (ask ~the fighter recall your
                                             time-of-departure))
                            (quotient (ask ~the fighter recall
                                           your speed) 3600.0)))
         else (if (not (equal (ask ~the fighter recall your status)
                              'destroyed))
                  then
                  (times (ask ~the fighter recall your fuel)
                         (ask ~the fighter recall your mpg)))))


(ask mathematician when receiving
     (velocity when going from >p1 to >p2 in >time)
     (list (quotient (difference (car p2) (car p1))
                     time)
           (quotient (difference (cadr p2) (cadr p1))
                     time)))


(ask mathematician when receiving
     (new position after traveling from >p at >v in >time)
     (list (plus (car p)
                 (times (quotient (car v) 3600) time))
           (plus (cadr p)
                 (times (quotient (cadr v) 3600) time))))
```

## 6.16. Physicist

```
(ask simulator make physicist
     with positions-blinded nil)


(tell physicist when receiving
  (>m megaton bombs exploded at >position altitude >h)
    (~block with local variables (blind-time blind-radius)
        (~let blind-radius be (times 50 (sqrt m) (expt h 0.7)))
        (~let blind-time   be (times 20 blind-radius))
        (foreach radar in (append (ask SAM recall your offspring)
                                  (ask GCI recall your offspring)
                                  (~every AWACS))
                 do
          (if (~you find ~the radar within ~the blind-radius of ~the position)
           then
              (~you deactivate ~the radar for ~the blind-time)))
        (~you add ~the position to your list of positions-blinded)
        (~you plan after ~the blind-time seconds remove ~the position
              from your list of positions-blinded)))
```

```
(tell physicist when receiving (find >GCI within >distance of >position)
    (< (ask mathematician distance from !(ask ~the GCI recall your position)
                                        to ~the position)
       distance))


(tell physicist when receiving (deactivate >GCI for >duration)
    (~you unplan all (tell scheduler ~the GCI is unblinded))
    (~you schedule after ~the duration seconds tell scheduler ~the GCI
      is unblinded)
    (if (eq (ask ~the GCI recall your status) 'active') then
        (tell scheduler ~the GCI is blinded)))
```

## 7.0. References

[FAU80] Faught, W. S., Klahr, P., and Martins, G. R. An Artificial Intelligence Approach To Large-Scale Simulation. Proceedings Summer Computer Simulation Conference, Seattle, 1980.

[FAU81] Faught, W. S. and Klahr, P. An Analysis of Proximity-Detection and Other Algorithms in the ROSS Simulator. N-1587-AF, The Rand Corporation, Santa Monica, 1981.

[GOL76] Goldberg, A. and Kay, A. Smalltalk-72 Instruction Manual. SSL 76-6, Xerox PARC, 1976.

[GOL81] Goldin, S. E. and Klahr, P. Learning and Abstraction in Simulation. Proceedings International Joint Conference on Artificial Intelligence, Vancouver, 1981.

[HEW77] Hewitt, C. Viewing Control Structures as Patterns of Message Passing. Artificial Intelligence, 8, 1977, 323-364.

[KAH79] Kahn, K. M. Director Guide. AI Memo 482B, MIT, 1979.

[KLA80a] Klahr, P. and Faught, W. S. Knowledge-Based Simulation. Proceedings First Annual National Conference on Artificial Intelligence, Palo Alto, 1980.

[KLA80b] Klahr, P., Faught, W. S. and Martins, G. R. Rule-Oriented Simulation. Proceedings 1980 IEEE Conference on Cybernetics and Society, Cambridge, Massachusetts, 1980.

[MCA81] McArthur, D. and Sowizral, H. An Object-Oriented Language for Constructing Simulations. Proceedings International Joint Conference on Artificial Intelligence, Vancouver, 1981.

[MCA82] McArthur, D. and Klahr, P. The ROSS Language Manual. N-1854-AF, The Rand Corporation, Santa Monica, 1982.

[STA81] Stallman, R. M. EMACS Manual for TWENEX Users. AI Memo 555, MIT, Cambridge, Massachusetts, 1981.